900 Series HP 3000 Computers

HP Symbolic Debugger/iX User's Guide

HEWLETT PACKARD

HP Part No. 31508-90003
Printed in U.S.A. June 1992

Third Edition
E0692

Print History

The following table lists the printings of this document, together with the respective release dates for each edition. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	**Date**
First Edition	October 1989
Second Edition	April 1990
Third Edition	June 1992

Preface

The *HP Symbolic Debugger/iX User's Guide* explains how to debug computer programs on 900 Series HP 3000 computer systems. The manual assumes that you are an experienced programmer familiar with symbolic debuggers on other systems.

This manual contains the following chapters:

Chapter 1	Introduces the HP Symbolic Debugger/iX - what it is and who can use it. This chapter also explains how to prepare a program for use with the symbolic debugger.
Chapter 2	Contains listings of sample debugger programs which are used in sample debugger sessions online. Use these listings for reference to the online programs when experimenting with the debugger.
Chapter 3	Describes how to use HP Symbolic Debugger to debug programs.
Chapter 4	Discusses the HP Symbolic Debugger commands.
Appendix A	Lists warning and error messages, along with their remedial actions.
Appendix B	Lists the language operators for HP C.
Appendix C	Lists the language operators for HP FORTRAN 77 and explains FORTRAN VMS record support.
Appendix D	Lists the language operators for HP Pascal.
Appendix E	Lists the language operators for HP COBOL II.
Appendix F	Lists special variables used by the HP Symbolic Debugger.
Appendix G	Lists some limitations of HP Symbolic Debugger and gives some usage hints.
Appendix H	Lists installed HP Symbolic Debugger files.
Appendix I	Lists the HP Symbolic Debugger commands.
Appendix J	Lists the registers displayed by the debugger in disassembly mode.
Glossary	Lists new terms and their definitions.

Additional Documentation

This manual does not discuss the MPE/iX operating system in detail. Only those aspects relevant to HP Symbolic Debugger are mentioned. Similarly, details about compiling a program using HP COBOL II, HP FORTRAN 77, HP Pascal, and HP C are only discussed to the extent that they affect how you use HP Symbolic Debugger. See the appropriate operating system or language manual for complete information about those subjects. The following is a partial list of the operating system and language manuals:

Manual Title	**Manual Part Number**	**Number to Use to Order Manual**
HP COBOL II/XL Reference Manual	31500-90001	31500-90001
HP COBOL II/XL Programmer's Guide	31500-90002	31500-90002
HP COBOL II/XL Quick Reference Guide	31500-90003	31500-90003
HP FORTRAN 77/iX Reference	31501-90010	31501-60021
HP FORTRAN 77/iX Programmer's Guide	31501-90011	31501-60022
HP FORTRAN 77/iX Migration Guide	31501-90004	31501-90023
HP Pascal/iX Reference Manual	31502-90001	31502-90001
HP Pascal/iX Programmer's Guide	31502-90002	31502-90002
HP C/iX Reference Manual	31506-90005	31506-90005
HP C/iX Library Reference Manual	30026-90001	30026-90001
HP C Programmer's Guide	92434-90002	92434-90002
HP Link Editor/iX Reference Manual	32650-90030	32650-90030
MPE/iX Commands Reference Manual	32650-90003	32650-60002
MPE/iX Intrinsics Manual	32650-90028	32650-90028
PA-RISC 1.1 Architecture and Instruction Set	09740-90039	09740-90039

Conventions

`CASE`	In a syntax statement, commands and keywords are shown in uppercase and lowercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:
	`SHOWJOB`
	can be entered as any of the following:
	`showjob` `Showjob` `SHOWJOB`
	It cannot, however, be entered as:
	`shojwob` `Shojob` `SHOW_JOB`

italics	In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file: `RELEASE` *filename* Italics font is also used to emphasize a *word* or *words*.
punctuation	In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered: (*filename*):(*filename*)
underlining	Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, "yes" is the user's response to the prompt: `Do you want to continue? >>` <u>`yes`</u>
`{ }`	In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `ON` or `OFF`: $\texttt{SETMSG} \left\{ \begin{array}{l} \texttt{ON} \\ \texttt{OFF} \end{array} \right\}$ Commands listed in braces are called *command lists* throughout this manual.

Conventions (continued)

`[ ]`

In a syntax statement, brackets enclose optional elements. In the following example, `,TEMP` can be omitted:

`PURGE` *filename*`[,TEMP]`

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select *devicename* or *deviceclass* or neither. The elements cannot be repeated.

$$\texttt{SHOWDEV} \begin{bmatrix} \textit{devicename} \\ \textit{deviceclass} \end{bmatrix}$$

`[ ... ]`

In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *itemname* zero or more times. Each instance of *itemname* must be preceded by a comma:

`[,`*itemname*`] [...]`

In the example below, you only use the comma as a delimiter if *itemname* is repeated; no comma is used before the first occurrence of *itemname*:

`[`*itemname*`] [,...]`

`| ... |`

In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select `A`, `AB`, `BA` or `B`. The elements cannot be repeated.

$$\begin{Bmatrix} \texttt{A} \\ \texttt{B} \end{Bmatrix} | \ldots |$$

... ⋮

In an example, horizontal or vertical ellipses indicate where portions of the example have been omitted.

`Δ`

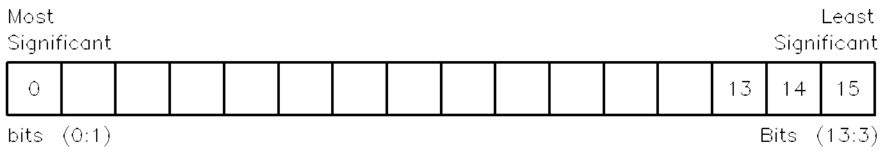
In a syntax statement, the space symbol `Δ` shows a required blank. In the following example, *modifier* and *variable* must be separated with a blank:

`SET [(`*modifier*`)]Δ(`*variable*`);`

( )	The symbol ( ) indicates a key on the keyboard. For example, (RETURN) represents the carriage return key.

Conventions (continued)

CNTL *char*	CNTL *char* indicates a control character. For example, CNTL Y means you press the control key and the Y key simultaneously.
Comment	Explains an operator entry or debug message.
>	The HP Symbolic Debugger prompt.
|	Represents "or".
;	Separates commands in a command list.
base prefixes	The prefixes %, #, and $ specify the numerical base of the value that follows: %*num* specifies an octal number #*num* specifies a decimal number $*num* specifies a hexadecimal number If no base is specified, decimal is assumed.
`Bits` (*bit:length*)	When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format `Bits` (*bit:length*) *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, `Bits (13:3)` indicates bits 13, 14, and 15:



Contents

Figures

Tables

1

Introducing HP Symbolic Debugger/iX

The HP Symbolic Debugger is an interactive tool that assists you in finding errors in programs written in high-level programming languages.

On most terminals, the HP Symbolic Debugger uses the full screen. The screen is divided into areas that let you view source code, commands that you enter and command and program output. When you work with the debugger, you use the same language constructs that are used in the program you're debugging.

The HP Symbolic Debugger lets you:

View source code	You can view any program source line readily.
Display and modify variables	You can view the value of any type of data item in the program and you can display it in the format that is most appropriate. When necessary, you can change the value of a data item.
Trace program flow	You can execute one or more statements at one time, allowing you to closely examine program flow and data areas. If the program is large, you might prefer to set breakpoints at certain statements in the program. When the breakpoints occur, you can examine data areas and alter them if necessary. If your program contains several procedure calls, you might want to display the program stack to trace those calls.
Capture and rerun a debugger session	If you think you might need to retrace your steps during a debugger session, you can have the debugger automatically record your session commands in a file. Then, at a later time, you can replay those commands. This playback feature can save you time because it contains the "trail" of commands that led to a given program state.
Execute debugger commands before each source statement	You can have the debugger execute one or more commands before it executes each source statement in the program. These commands, called assertions, can save you time when you need to examine execution progress one line at a time.

Who Can Use HP Symbolic Debugger

HP Symbolic Debugger can be used by programmers who program in HP COBOL II, HP C, HP Pascal, and HP FORTRAN 77.

Creating a Program with Debugger Information

To debug a program on the symbolic level, you must compile and link the source program with debugger information to create an executable program file. Figure 1-1 illustrates the process of creating an executable program file containing debugger information. If you do not compile and link your program with debugger information, the debugger can only display the register values, absolute addresses, and labels.

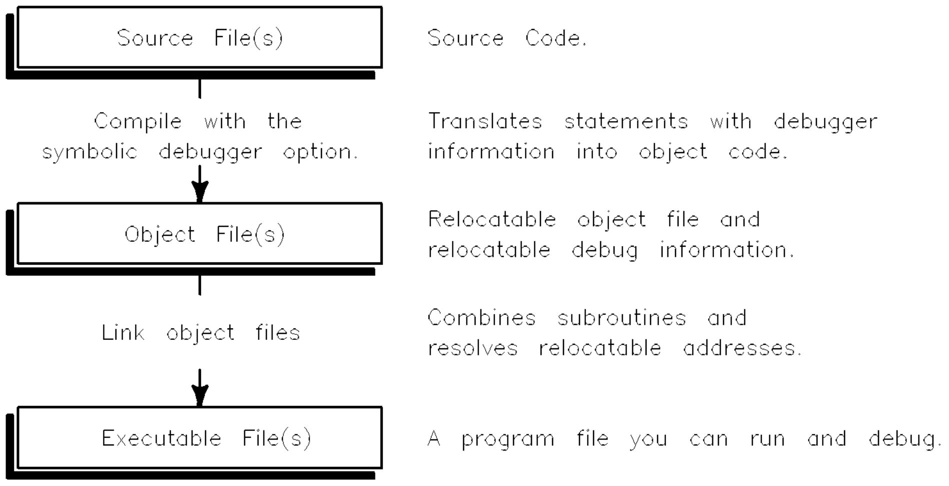

Source File(s)
Source Code.
Compile with the symbolic debugger option.
Translates statements with debugger information into object code.
Object File(s)
Relocatable object file and relocatable debug information.
Link object files
Combines subroutines and resolves relocatable addresses.
Executable File(s)
A program file you can run and debug.

Figure 1-1. Creating an Executable Program File

Terminal Support

Only Hewlett-Packard terminals with memory lock support the command windows used by the debugger. Non—Hewlett-Packard terminals or Hewlett-Packard terminals without memory lock operate in line mode only. Use the *-L* command line option when invoking the debugger to operate in line mode.

Command History

The symbolic debugger has a command history mechanism modeled after MPE/iX's command interpreter. The symbolic debugger recognizes the following history commands:

```
do

redo

listredo
```

The differences between MPE/iX's command interpreter and the symbolic debugger's command history mechanism are as follows:

- The symbolic debugger has a default redo stack size of 20.
- The redo stack size is set to whatever size you want by changing the value of the variable XDBREDOSIZE before invoking the symbolic debugger.
- The symbolic debugger only uses lower case editing directives.

Where To Go from Here

You are now familiar with the major features of HP Symbolic Debugger.

To get hands-on practice in running the debugger, continue on with the next chapter. It tells you how to run a sample debugger session.

If you don't have time to run the sample session, but want to start debugging a program right away, skip to chapter 3. Chapter 3 introduces you to the most common ways to use the debugger and should give you enough information to begin using it.

Use chapter 4 as a reference chapter. It lists details about each of the HP Symbolic Debugger commands.

See appendix A for error message information.

See appendix B to find out the language operators for HP C.

See appendix C to find out the language operators for HP FORTRAN 77 and VMS FORTRAN record support.

See appendix D to find out the language operators for HP Pascal.

See appendix E to find out the language operators for HP COBOL II.

See appendix F for a list of the special variable used by the HP Symbolic Debugger.

See appendix G for a list of limitations and hints.

See appendix H for a list of installed files for HP Symbolic Debugger.

See appendix I for a list of the HP Symbolic Debugger commands.

See appendix J for a list of registers displayed by HP Symbolic Debugger in disassembly mode.

See the glossary for definitions of new terms.

2

Getting Started

HP Symbolic Debugger/iX comes with a sample debugger session for each of the supported native mode languages, HP COBOL II, HP FORTRAN 77, HP Pascal, and HP C. You can run it without knowing anything about the debugger; the debugger guides you through each step. The session takes only a few minutes to run. When you're finished, you will have a good overview of how the debugger works and some important ways it can be used.

When running the sample session, follow the instructions explained at the beginning of the session. The program used in the debugger session is listed at the end of this chapter.

The Debugger Session Scenario

The following explains the scenario under which the sample debugger session runs.

You're developing a program to read and process rainfall data. Proceeding in stages, you're developing the user input section and the portion that fills in an array with data from the rainfall file.

During tests, you've encountered program aborts with messages indicating that access to memory outside your program's allotment has occurred. This type of error most frequently results from bad pointer arithmetic or bad array subscripts, especially in a loop. This program does no explicit pointer arithmetic, so you've decided to use HP Symbolic Debugger to check the loops in your program.

Figure 2-1. Sample Debugger Session Scenario

Running the Sample Session

To run the sample debugger session, log on to your account. Then enter the command below that corresponds to the language you're most familiar with:

HP COBOL II—>	`xdb;info="-d pub.sys democbx.pub.sys"`
HP FORTRAN 77—>	`xdb;info="-d pub.sys demofx.pub.sys"`
HP Pascal—>	`xdb;info="-d pub.sys demopx.pub.sys"`
HP C—>	`xdb;info="-d pub.sys democx.pub.sys"`

This will get you inside HP Symbolic Debugger. You used the *-d* option to specify the directory where the source file is. (This corresponds to the *D* (*Directory*) command which you can use inside the debugger for the same purpose).

Now to run the sample session, type the command below that corresponds to your language:

HP COBOL II—>	`<<democbxr.pub.sys`
HP FORTRAN 77—>	`<<demofxr.pub.sys`
HP Pascal—>	`<<demopxr.pub.sys`
HP C—>	`<<democxr.pub.sys`

You will now be in the debugger session. Follow the instructions on the screen to continue. When the session ends, you will still be in the debugger. To get out, enter:

```
>q
```

Note

The sample session is really a playback file which was created with the debugger. If you want to know more about the Record and Playback Facility of the debugger, refer to the section "Record and Playback Commands" in Chapter 4 "HP Symbolic Debugger Commands".

Where To Go from Here

Now that you've completed the sample session, you have a good idea about how HP Symbolic Debugger works. To learn more details about the operations used in the debugger session or to begin debugging your own programs, continue with chapter 3. If you want to see the listing for the program you saw in the session, read on.

Sample Program Listings

This section lists the language source files used in the sample debugger sessions on line.

These source files:	are listed in:
`DEMOCBMS`	Figure 2-2
`DEMOCBSS`	Figure 2-3
`DEMOFS`	Figure 2-4
`DEMOPS`	Figure 2-5
`DEMOCS`	Figure 2-6

Sample HP COBOL II Program

```
$CONTROL BOUNDS, VERBS, MAP, CROSSREF
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  RAIN-REPORT.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
   77  NUMBER-YEARS              PIC 9.
   77  FIRST-YEAR                PIC 9(4).
   01  YEAR-INDEX                PIC 9(4)    COMP.
   01  NUM-OF-MONTHS             PIC 9(4)    COMP.
   01  MONTH-TOTALS              VALUE ZEROS.
       05  MT-TABLE              PIC 9(6)V99
                                 OCCURS 60 TIMES.
 PROCEDURE DIVISION.
 100-MAIN-ROUTINE.
     PERFORM 200-GET-INPUT.
     PERFORM 300-CALCULATE.
     PERFORM 400-PRINT-REPORT.
     PERFORM 500-STOP-RUN.
 200-GET-INPUT.
     DISPLAY "ENTER THE FIRST YEAR YOU WISH TO REPORT ON:".
     ACCEPT FIRST-YEAR.
     IF (FIRST-YEAR < 1950) OR (1988 < FIRST-YEAR) THEN
         GO 200-GET-INPUT.
     DISPLAY "ENTER THE # OF YEARS YOU WISH TO CONSIDER (1-5):".
     ACCEPT NUMBER-YEARS.
     IF (NUMBER-YEARS < 0) OR (5 < NUMBER-YEARS) THEN
         GO 200-GET-INPUT.
 300-CALCULATE.
     COMPUTE YEAR-INDEX = (FIRST-YEAR - 1950) * 12.
     COMPUTE NUM-OF-MONTHS = NUMBER-YEARS * 122.
     CALL "LOADMT" USING YEAR-INDEX,
                         NUM-OF-MONTHS,
                         MONTH-TOTALS.
 400-PRINT-REPORT.
 500-STOP-RUN.
     STOP RUN.
```

Figure 2-2. HP COBOL II Main Source File, DEMOCBMS

Sample HP COBOL II Program

```
$CONTROL SUBPROGRAM, BOUNDS, VERBS, MAP, CROSSREF
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  LOADMT.
 AUTHOR.  ANON.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT RAINFILE ASSIGN TO "RAINFALL.pub.sys".
 DATA DIVISION.
 FILE SECTION.
   FD RAINFILE.
   01  INCHES-PER-MONTH          PIC 9(6)V99.
 WORKING-STORAGE SECTION.
   77  77-INDEX                  PIC S9(4)    COMP.
 LINKAGE SECTION.
   01  YEAR-INDEX                PIC 9(4)     COMP.
   01  NUM-OF-MONTHS             PIC 9(4)     COMP.
   01  MONTH-TOTALS.
       05  MT-TABLE              PIC 9(6)V99  COMP
                                 OCCURS 60 TIMES.
 PROCEDURE DIVISION USING YEAR-INDEX,
                          NUM-OF-MONTHS,
                          MONTH-TOTALS.
 100-MAIN-SUBROUTINE.
     OPEN INPUT RAINFILE.
     PERFORM 200-READ-RAINFILE YEAR-INDEX TIMES.
     MOVE 1 TO 77-INDEX.
     PERFORM 300-LOAD-MT-TABLE NUM-OF-MONTHS TIMES.
     CLOSE RAINFILE.
     PERFORM 500-GOBACK.
 200-READ-RAINFILE.
     READ RAINFILE AT END GO 500-GOBACK.
 300-LOAD-MT-TABLE.
     PERFORM 200-READ-RAINFILE.
     MOVE INCHES-PER-MONTH TO MT-TABLE (77-INDEX).
     ADD 1 TO 77-INDEX.
 500-GOBACK.
     GOBACK.
```

Figure 2-3. HP COBOL II Subroutine Source File, DEMOCBSS

```
$CONTROL RANGE, CODE_OFFSETS, TABLES
      PROGRAM RAIN_REPORT
      INTEGER*2 NUMBER_YEARS,
     2          FIRST_YEAR,
     3          YEAR_INDEX,
     4          NUM_OF_MONTHS
      REAL MONTH_TOTALS(60)
  100 PRINT *,'ENTER THE FIRST YEAR YOU WISH TO REPORT ON:  '
      READ (5,*) FIRST_YEAR
      IF ((FIRST_YEAR .LT. 1950).OR.(FIRST_YEAR .GT. 1988)) THEN
        GOTO 100
      ENDIF
  110 PRINT *,'ENTER THE # OF YEARS YOU WISH TO CONSIDER (1-5):  '
      READ (5,*) NUMBER_YEARS
      IF ((NUMBER_YEARS .LT. 1).OR.(NUMBER_YEARS .GT. 5)) THEN
        GOTO 110
      ENDIF
      YEAR_INDEX = (FIRST_YEAR - 1950) * 12
      NUM_OF_MONTHS = NUMBER_YEARS * 122
      CALL LOADMT (YEAR_INDEX, NUM_OF_MONTHS, MONTH_TOTALS)
      PRINT *,'PROGRAM ENDS'
      STOP
      END

      SUBROUTINE LOADMT (YEAR_INDEX, NUM_OF_MONTHS, MONTH_TOTALS)
      INTEGER*2 YEAR_INDEX,
     2          NUM_OF_MONTHS,
     3          TABLE_INDEX
      REAL MONTH_TOTALS(60),
     2     HOLD_RAINFALL
      OPEN (UNIT=10, FILE='RAINFALL.pub.sys')
      DO I=1, YEAR_INDEX
        READ (10,*) HOLD_RAINFALL
      END DO
      DO TABLE_INDEX = 1,NUM_OF_MONTHS
        READ (UNIT=10, FMT=10, END=900) HOLD_RAINFALL
        MONTH_TOTALS(TABLE_INDEX) = HOLD_RAINFALL
      END DO
  900 RETURN
      END
```

Figure 2-4. HP FORTRAN 77 Main Source File, DEMOFS

Sample HP Pascal Program

```
$RANGE ON, CODE_OFFSETS ON, TABLES ON$
program RainReport (INPUT, OUTPUT, RainFall);

type
  YearType       = 1900..2000;
  NumYearsType   = 0..200;
  MonthTotalType = REAL;
  ArrayType      = ARRAY [1..60] of MonthTotalType;

var
  NumYears     :  NumYearsType;
  FirstYear    :  YearType;
  YearIndex    :  INTEGER;
  NumOfMonths  :  INTEGER;
  MonthTable   :  ArrayType;
  RainFall     :  TEXT;

procedure GetInput;
{
This procedure prompts the user for the initial year and number of
years for the report.  It also checks to see that the year and number
of years are within range.
}
const
  YearPrompt     = 'Enter the first year on which to report:  ';
  NumYearsPrompt = 'Enter the # of years to consider (1 - 5):  ';

procedure GetFirstYear;
begin {GetFirstYear statements};
  writeln (OUTPUT);
  prompt (OUTPUT, YearPrompt);
  readln (INPUT, FirstYear);
  IF (FirstYear < 1950) or (FirstYear > 1988) THEN
    GetFirstYear;
end {GetFirstYear statements};
```

Figure 2-5. HP Pascal Main Source File, DEMOPS

```
procedure GetNumYears;
begin {procedure GetNumYears statements};
  writeln (OUTPUT);
  prompt (OUTPUT, NumYearsPrompt);
  readln (INPUT, NumYears);
  IF (NumYears < 1) or (NumYears > 5) THEN
    GetNumYears;
end;

begin {level 1 procedure};
  GetFirstYear;
  GetNumYears;
  YearIndex   := (FirstYear - 1950) * 12;
  NumOfMonths := NumYears * 122;
end {level 1 procedure};

procedure LoadMonthTable;
var
  ArrayIndex    :  INTEGER;
  HoldRainFall  :  INTEGER;

begin {LoadMonthTable statements};
  HoldRainFall := 0;
  reset (RainFall, 'RAINFALL.pub.sys');
  FOR ArrayIndex := 1 to YearIndex DO
{
    This loop will perform dummy reads to get the file to the start
    of the requested data.
}
    readln (RainFall, HoldRainFall);
  FOR ArrayIndex := 1 to NumOfMonths DO
    begin {FOR loop}
      readln (RainFall, HoldRainFall);
      MonthTable[ArrayIndex] := HoldRainFall / 100
    end {FOR loop}
end {LoadMonthTable statements};

begin {main program}
  GetInput;
  LoadMonthTable
end {of program}.
```

Figure 2-5. Pascal Main Source File, DEMOPS (Continued)

```
#include <stdio.h>

#define YEAR_PROMPT       "\nEnter the first year on which to report:  "
#define NUM_YEARS_PROMPT  "\nEnter the # of years to consider (1 - 5):  "

typedef  int               year_type;
typedef  int               num_years_type;
typedef  double            month_total_type;
typedef  month_total_type  array_type[60];

num_years_type    num_years;
year_type         first_year;
int               year_index;
int               num_of_months;
array_type        month_table;
FILE             *rain_fall,
                 *fopen();

void get_first_year()
{
   printf (YEAR_PROMPT);
   scanf ("%d", &first_year);
   if ((first_year < 1950) || (first_year > 1988))
      get_first_year();
}

void get_num_years()
{
   printf (NUM_YEARS_PROMPT);
   scanf ("%d", &num_years);
   if ((num_years < 1) || (num_years > 5))
      get_num_years();
}
```

Figure 2-6. C Main Source File, DEMOCS

```
void get_input()
{
   /*
    * This function prompts the user for the initial year and number of
    * years for the report.  It also checks to see that the year and number
    * of years are within range.
    */
  get_first_year();
  get_num_years();
  year_index    = (first_year - 1950) * 12;
  num_of_months = num_years * 122;
}

void load_month_table()
{
   int  array_index;
   int  hold_rain_fall = 0;

   rain_fall = fopen("RAINFALL.pub.sys", "r");
   /* This loop will perform dummy reads to get the file to the start
    * of the requested data.
    */
   for (array_index = 1; array_index <= year_index; array_index++)
      fscanf (rain_fall, "%d", &hold_rain_fall);
   for (array_index = 1; array_index <= num_of_months; array_index++) {
      fscanf (rain_fall, "%d", &hold_rain_fall);
      month_table[array_index] = hold_rain_fall / 100;
   }
}

main()
{
   get_input();
   load_month_table();
}
```

Figure 2-6. C Main Source File, DEMOCS (Continued)

3

Using the HP Symbolic Debugger

This chapter shows you how to start HP Symbolic debugger and how to use its major features. The first sections of the chapter list the steps you must perform to begin using the debugger and familiarize you with the screen display. The last sections of the chapter show you how to perform various tasks. You do not perform these tasks necessarily in the same order as they are listed; pick and choose the tasks depending on your requirements.

To get started with HP Symbolic Debugger, read and perform these sections in order:

- Preparing the Program
- Starting the HP Symbolic Debugger
- Starting the Program

Once you start the program, read and perform the sections below that correspond to the tasks you need to perform:

- Ending the Program
- Ending the HP Symbolic Debugger
- Displaying Lines in the Source Program
- Controlling the Command Window Display
- Changing the Source Window Size
- Displaying Assembly Code
- Displaying Source and Assembly Code
- Stepping through the Program
- Searching for a String in the Program
- Pausing during Execution
- Displaying Data
- Modifying Data
- Tracing Function and Procedure Calls
- Capturing and Rerunning a Debug Session
- Executing Commands at Each Source Line
- Using Macros
- Altering the Execution Sequence
- Getting Help

Preparing the Program

Before starting HP Symbolic Debugger, compile your HP COBOL II, HP FORTRAN 77, HP Pascal, or HP C program using the symbolic debugger option. If you do not use the symbolic debugger option, you can only debug the program in disassembly mode; the debugger can track only register values, absolute addresses and labels.

When you're confident that the program will compile without errors, use the symbolic debugger compile option. When you use the symbolic debugger option, the compiler generates tables containing the names and addresses of variables, labels and source lines. These tables are the symbolic hooks into your program.

There are two ways to compile your program with symbolic debugger information:

- Embed the symbolic debugger option in the first statement of your source code.
- Use the *info-string* option to specify the symbolic debugger option when you compile your program.

Table 3-1 lists the compiler embedded and info string options you can use to prepare programs for use with HP Symbolic Debugger.

Table 3-1. Compiler Embedded and Info-string Options

Compiler	Source Language Option	Info-string Argument
HP COBOL II	$control symdebug = xdb	Info = "$control symdebug = xdb"
HP FORTRAN 77	$symdebug xdb|off	Info = "symdebug xdb|off"
HP Pascal	$symdebug 'xdb'$	Info = "symdebug 'xdb'"
HP C	*None*	Info ="-g"

When compiling and linking, you need to link the file *xdbend.lib.sys* with the object files of your program. Specify *xdbend.lib.sys* as the last file in your link list. *xdbend.lib.sys* provides space in the user process for the debugger and is required.

The example below compiles and links the HP C program *test1c* producing the executable object program *TEST1P*:

```
:CCXL TEST1C, TEST1O;INFO="-g"
:LINK FROM=TEST1O, XDBEND.LIB.SYS; TO=TEST1P; RL=LIBCINIT.LIB.SYS
```

For HP COBOL II programs only, HP Symbolic Debugger uses the listing file *cobxdb* rather than the source files for the source viewing inside the debugger. Therefore, when you are debugging HP COBOL II programs derived from more than one source file, you must equate the default listing file *cobxdb* to a permanent listing file name before you compile each source file. You do not link the listing files with the object files.

For example, suppose you have an HP COBOL II program comprised of the source files *cob1* and *cob2*. When you compile each file, use the *xdb=* parameter to name your listing file. The following example compiles *cob1* into the object file *cob1o*, naming the listing file *lcob1*.

```
:COB85XL COB1, COB1O; XDB=LCOB1
```

Do the same with each additional program file.

```
:COB85XL COB2, COB2O; XDB=LCOB2
```

Alternatively, you can use the MPE *file* command to equate *cobxdb* to the listing file name. For example, to compile the same program files using this method, start by equating *cobxdb* to the listing file name *lcob1*.

```
:FILE COBXDB=LCOB1;SAVE
```

Then, compile *cob1* into the object file named *cob1o*. After compiling, you should be able to see the file *lcob1* when you do a *listf*.

```
:COB85XL COB1, COB1O
```

When you are ready to compile the second source file, change the *cobxdb* file equation:

```
:FILE COBXDB=LCOB2;SAVE
```

Then, compile *cob2* into an object file named *cob2o*. Again, you should be able to see the file *lcob2* when you perform a *listf*.

```
:COB85XL COB2, COB2O
```

Repeat this procedure for each source file comprising the program.

Once you have compiled all your source files by using one of the two previous methods, link the object files and the *xdbend.lib.sys* file together into an executable file named *cobx*:

```
:LINK FROM=COB1O, COB2O, XDBEND.LIB.SYS;TO=COBX
```

The debugger information in the object files know about the listing files *lcob1* and *lcob2* so you do not need to link these files. Once you've linked your program, you are ready to start HP Symbolic Debugger.

Starting the HP Symbolic Debugger

When using HP Symbolic Debugger, the debugger is the parent process and the program that you're debugging becomes a child process. The debugger controls only the child process and can debug only one child process at a time.

You can use HP Symbolic Debugger with sharable code, but you should be the only person using it at one time. (If someone else is using it also, they will encounter the same breakpoints, for example.)

Enter the following command to start the debugger:

```
                     ┌          ┌-d group[.acct]┐           ┐
                     │          │-r file        │           │
run xdb.pub.sys      │ ;info="  │-p file        │  [ ... ]" │
                     │          │-L             │           │
                     │          │-S num         │           │
                     └          └objectfile     ┘           ┘
```

You can run the debugger without options by entering:

`xdb` *objectfile*

You can also just type `xdb` and you will be prompted for the *objectfile*.

The HP Symbolic Debugger options are described below:

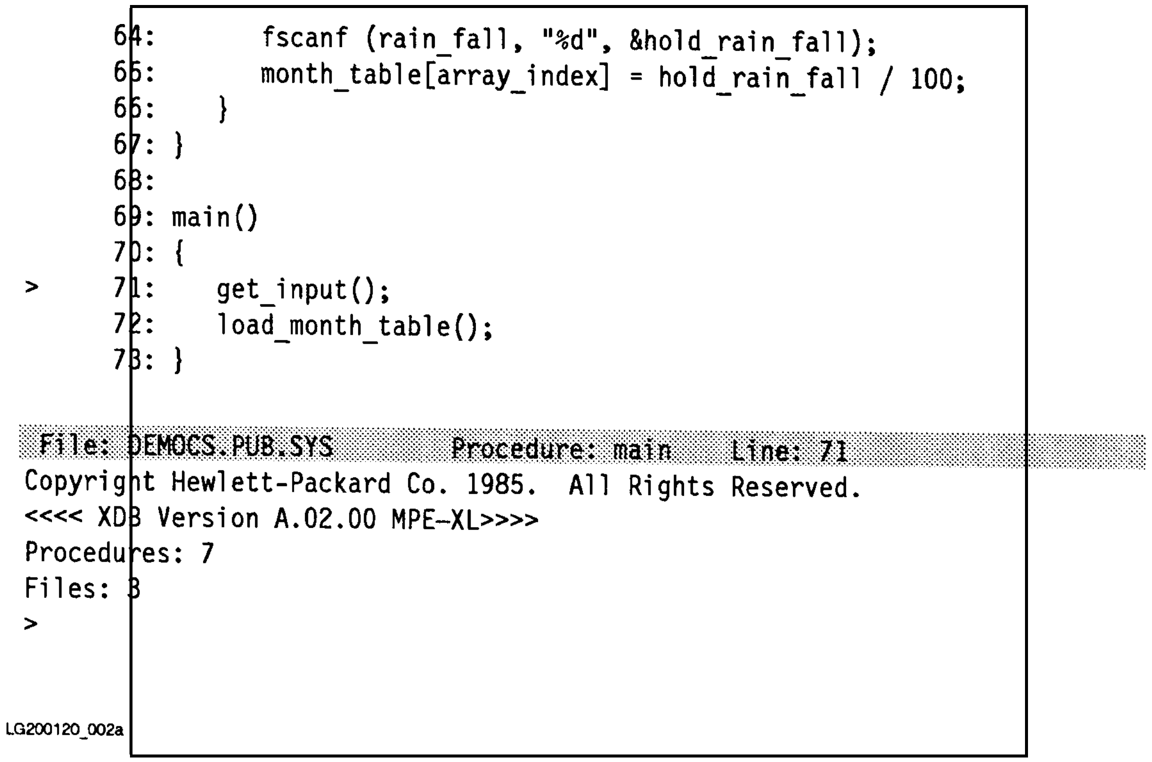
`-d` *group.[acct]*	This option names an alternate group and (optional) account containing the source files used to create the *objectfile*. Group and accounts are searched in the order that you list them. The current group and account is used if the file is not found in the group and account that you enter here. You can enter more than one `-d` option.
`-p` *file*	This option names a playback file created in a previous debugger session (see the `-r` option) or one that you created yourself.
`-r` *file*	This option names the file to which all debugger commands that you enter are recorded. You can use this file as a playback file in subsequent debug sessions (see the `-p` parameter). Recording begins as soon as you start the debugger. Any previous contents of the file are overwritten (no appending takes place).
`-L`	This option allows you to use the debugger in line mode when you do not have a terminal that supports memory lock.
`-S` *num*	This option sets the string-cache size to the number of bytes specified. The string cache holds data read from the *objectfile*. The default is 1024 bytes (1kb). Increasing the string cache size can improve debugger performance for large programs.
objectfile	This argument names the file that contains the executable code for the program. If you do not enter this option, you will be prompted for the *objectfile*. If this is the first time you are running the debugger, *objectfile* will be preprocessed to allow faster debugger startups in subsequent sessions.

Note

Equivalent debugger commands exist for the *-d*, *-p*, and *-r* options. See the *dir* (*directory*) and the "Record and Playback Commands" section in chapter 4, HP Symbolic Debugger Commands.

Once You Start HP Symbolic Debugger . . .

When you start HP Symbolic Debugger from a terminal that supports windowing, you see a source window similar to the one shown in Figure 3-1. If this is a large program and it is the first time you've run it under the debugger, it might take a few moments for the screen to appear. (The debugger preprocesses a program the first time it is run and displays the screen in less time during subsequent debugger sessions.)



```
         64:       fscanf (rain_fall, "%d", &hold_rain_fall);
         65:       month_table[array_index] = hold_rain_fall / 100;
         66:    }
         67: }
         68:
         69: main()
         70: {
>        71:    get_input();
         72:    load_month_table();
         73: }

 File: DEMOCS.PUB.SYS         Procedure: main    Line: 71
Copyright Hewlett-Packard Co. 1985.  All Rights Reserved.
<<<< XDB Version A.02.00 MPE-XL>>>>
Procedures: 7
Files: 3
>
```

LG200120_002a

Figure 3-1. The HP Symbolic Debugger Screen (Source Mode)

Note

The above screen appears only on terminals that allow memory locking. If your terminal does not have memory locking, the debugger displays information one line at a time (line mode).

The screen has three parts, which are described below. This is the screen you see when debugging in symbolic (*source*) mode.

Source window	The source window is located at the top of the screen, above the highlighted line. This is the area where you view the source statements. If your terminal has 24 lines, the top 15 are used for the source window. To alter the number of lines in the source window, see the section "Changing the Source Window Size" in this chapter. Source statements are displayed one window at a time. See the section "Displaying Lines in the Source Program" for directions on locating and displaying lines in the source window. The `>` prompt in the margin of the source window points to the current line. When you first start the debugger, this is the first executable statement. At other times, it is the line where the debugger is currently paused. Note that the source window is not limited to viewing the current line, and the `>` prompt may not always be visible.
Location window	The location window (or location line) is the highlighted line near the middle of the screen. This line shows you the current program file and procedure names and the source line number of the current line (the line currently being viewed in the source window).
Command window	The command window is the area located below the location window (highlighted line). This window is where the debugger commands that you enter are displayed. The debugger shows its own output in this area. The command window also shows output from the child process (program being debugged) The window automatically scrolls up when full, but this does not affect the other windows. A scrolling `more` feature lets you view debugger output on window-full at a time. The debugger prompts you to enter a command by displaying `>`. When you enter a command, enter the entire command on one line (continuation lines are not allowed). For information about controlling the display of lines in the command window, see the section "Controlling the Command Window Display."

At this point, before starting program execution, you might want to set breakpoints in the program, or change the source window size. The remaining sections in this chapter describe how you can accomplish these tasks and others as well (the tasks can also be performed during any execution pause). The sections are not listed in any particular order. You need to determine which are relevant to the debugging session at hand and perform only those.

Starting the Program

Once you start the debugger and you are ready to begin debugging your program, enter either an *r* (*run*), *R* (*Run*), *s* (*step*) or *S* (*Step*) command. The *r* (*run*) command starts execution of the program and allows you to enter arguments with it. The *R* (*Run*) command, as shown below, starts executing the program, but does not allow you to enter run-time arguments:

```
>R
```

To execute one statement (or one step of a Pascal statement) at a time, enter either the *s* (*step*) or *S* (*Step*) command. The initial *step* command executes the first statement of the program. The following *s* (*step*) command allows single-stepping through the program and any procedures that it contains:

```
>s
```

The following *S* (*Step*) command allows single-stepping through the program, stepping over procedure calls—A procedure call is treated as a single statement.

```
>S
```

Note

HP Symbolic Debugger commands are case sensitive; you must type them exactly as documented. To see command syntax, refer to each command's listing in chapter 4 "HP Symbolic Debugger Commands".

Ending the Program

If you want to terminate your program before it normally completes, enter the *k* (*kill*) command:

```
>k
```

You will be prompted to confirm this request. To have the debugger ignore the request,
enter `n`; otherwise, enter `y`.

At this time, you can restart the program, quit the debugger, or enter other commands.

Ending the HP Symbolic Debugger

To end HP Symbolic Debugger, enter the *q* (*quit*) command:

```
>q
```

You will be prompted to confirm this request. To have the debugger ignore the request,
enter `n`; otherwise, enter `y`.

Displaying Lines in the Program

There are several ways to display program lines in the source program window.

To display a particular source line, enter the *v* (*view*) command with the line number. For example, to display line 11:

```
>v 11
```

To move one or more lines forward in the program, enter the plus sign (+) and the number of lines you want to move. When moving forward or backward in the program, the source and location windows are adjusted accordingly. For example, to move five lines forward, enter:

```
>+5
```

To move backward in the program, enter the minus sign (-) and the number of lines you want to move. To move backwards five lines, enter:

```
>-5
```

Note

When you reach the end (or beginning) of the source program using the + and - commands, no further movement may take place.

You can repeat a previous + or - command (see *+5* and *-5* above) by pressing RETURN.

To display a procedure that has been called but is currently suspended at a given depth in the run-time stack, enter the *V (View)* command. The following example displays the procedure at depth two in the run-time stack. (Stack depth one is the current procedure's caller, depth two is its caller, etc.)

```
>V 2
```

To view the current point of suspension during program execution in the source window, use the *V* (*View*) command with no arguments:

```
>V
```

Note

The source window automatically tracks where the program becomes suspended, and the *V* (*View*) is only needed after using the *v* (*view*), +, or - commands.

Controlling the Command Window Display

Command and program output is displayed one screen at a time in the command window. You can use the terminal keys *PgUp*, *PgDn*, *Home*, *End* and the CTRL *arrow* keys (or the equivalent scroll keys on your terminal) to scroll the command window. When you enter a command that requires more than the number of lines in the command window to display, the debugger displays enough lines to fill the command window then displays `--More--`at the bottom.

Use one of the following commands to continue.

SPACE BAR	Displays one more window-full.
RETURN	Displays one more line.
`q`	Quits scrolling and ignores the rest of the output until another debugger prompt is issued.

To view command window output in a continuous stream, use the *sm* (*suspend more*) command to suspend the `more` feature. CNTL S may be used to temporarily suspend scrolling when the `more` feature is suspended. Use CNTL Q to continue scrolling.

To return to single-window output, enter the *am* (*activate more*) command.

Note

Output from the child process (program being debugged) also appears in the command window, but it is *not* controlled by the `more` feature.

Changing the Source Window Size

To change the size of the source window, use the *w* (*window*) command and specify the number of lines you want for this window. For example, to change the size of the source window to 12 lines enter:

```
>w 12
```

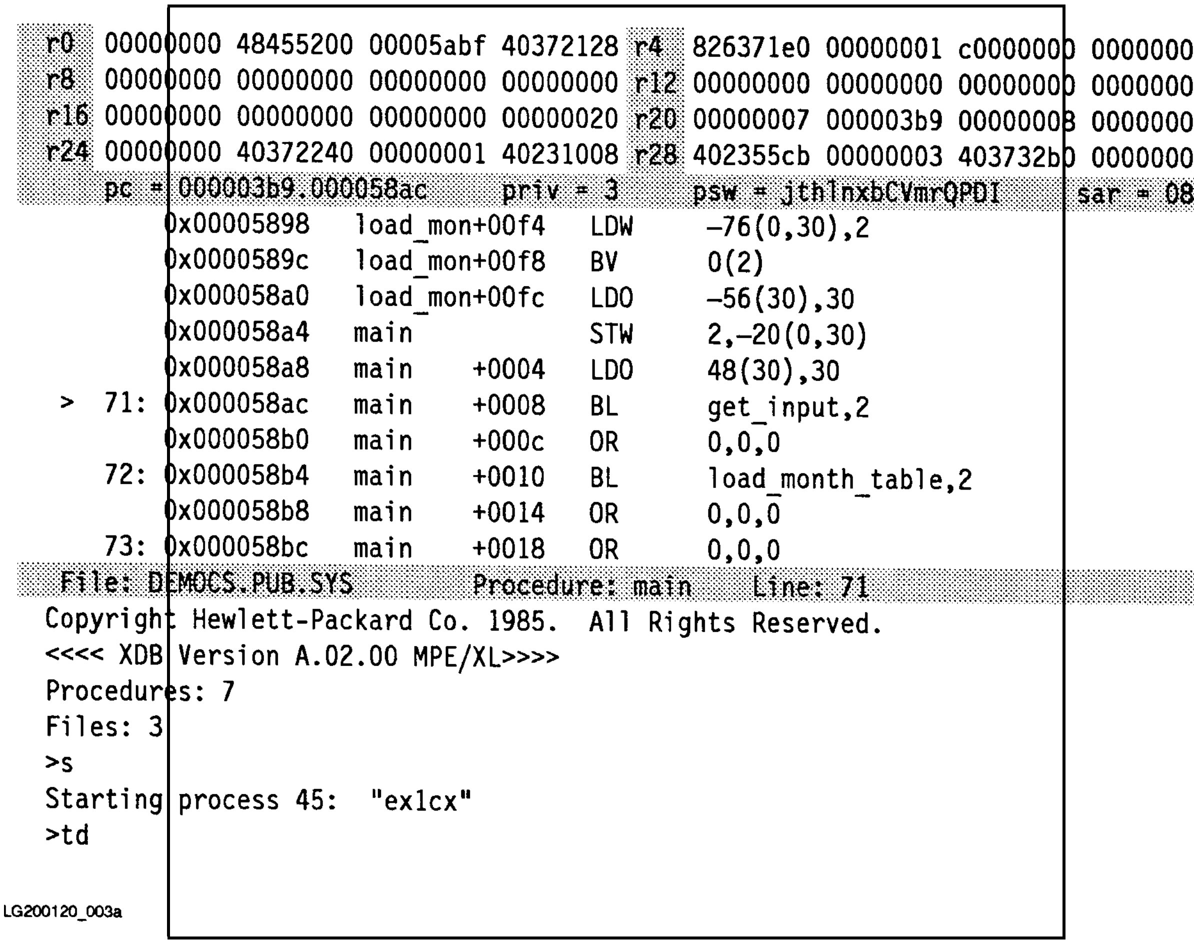
The number of lines for the source window range from one to 21 for a 24-line terminal (the default is 15). Changing the size of the source window also changes the size of the command window.

Displaying Assembly Code

If you didn't use the symbolic debugger option when compiling the program, you will be debugging in disassembly mode and will see a screen similar to the one shown in Figure 3-2. Even if you compiled with the symbolic debugger option, you can debug in disassembly mode by entering the *td* (*toggle disassembly*) command as follows:

```
>td
```

In disassembly mode, the program is debuggable at the machine instruction level. Note that corresponding source-line numbers are displayed along with the absolute and symbolic address of each instruction. The values of all hardware registers are also shown in disassembly mode. A highlighted register value indicates its contents were modified by the last instruction executed.



```
 r0  00000000 48455200 00005abf 40372128 r4  826371e0 00000001 c0000000 0000000
 r8  00000000 00000000 00000000 00000000 r12 00000000 00000000 00000000 0000000
 r16 00000000 00000000 00000000 00000020 r20 00000007 000003b9 00000008 0000000
 r24 00000000 40372240 00000001 40231008 r28 402355cb 00000003 403732b0 0000000
     pc = 000003b9.000058ac     priv = 3    psw = jthlnxbCVmrQPDI     sar = 08
         0x00005898   load_mon+00f4  LDW    -76(0,30),2
         0x0000589c   load_mon+00f8  BV     0(2)
         0x000058a0   load_mon+00fc  LDO    -56(30),30
         0x000058a4   main           STW    2,-20(0,30)
         0x000058a8   main    +0004  LDO    48(30),30
 >   71: 0x000058ac   main    +0008  BL     get_input,2
         0x000058b0   main    +000c  OR     0,0,0
     72: 0x000058b4   main    +0010  BL     load_month_table,2
         0x000058b8   main    +0014  OR     0,0,0
     73: 0x000058bc   main    +0018  OR     0,0,0
  File: DEMOCS.PUB.SYS          Procedure: main     Line: 71
Copyright Hewlett-Packard Co. 1985.  All Rights Reserved.
<<<< XDB Version A.02.00 MPE/XL>>>>
Procedures: 7
Files: 3
>s
Starting process 45:  "ex1cx"
>td
```

LG200120_003a

Figure 3-2. The HP Symbolic Debugger Screen (Disassembly Mode)

To return to source mode, enter `td` again.

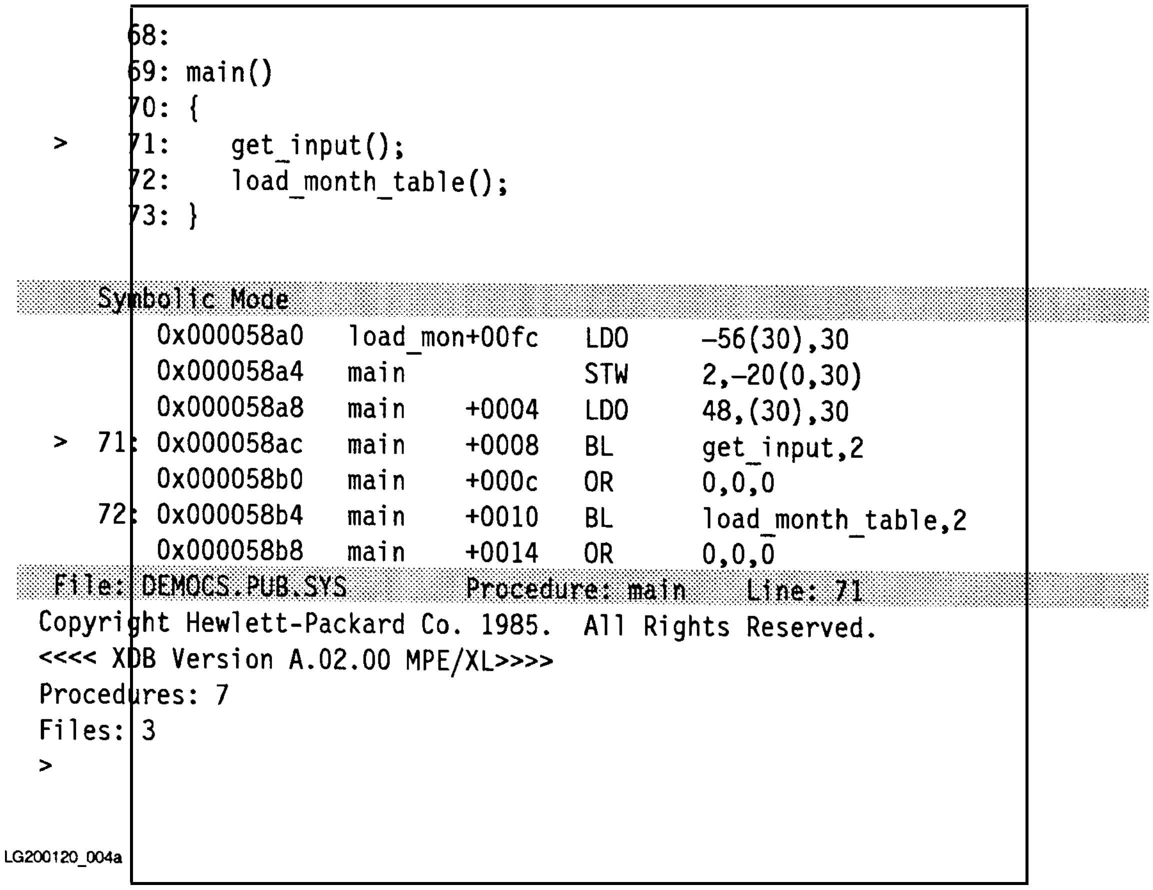
Disassembly mode can also be used when only parts of your program were compiled with the symbolic debugger option. Libraries linked in with your program are generally not debuggable unless disassembly mode is used, regardless of whether your program was compiled with the symbolic debugger option. Refer to appendix F “Registers Displayed by HP Symbolic Debugger” to see the registers displayed by the debugger in disassembly mode.

Displaying Source and Assembly Code

To view both the source code and its matching assembly code, enter the *ts* (*toggle screen*) command. When you do this, the source window is divided into two windows, the top for source code and the bottom for assembly code as shown in Figure 3-3.

To view source and assembly code, enter:

```
>ts
```



```
        68:
        69: main()
        70: {
>       71:    get_input();
        72:    load_month_table();
        73: }

      Symbolic Mode
          0x000058a0  load_mon+00fc   LDO     -56(30),30
          0x000058a4  main            STW     2,-20(0,30)
          0x000058a8  main    +0004   LDO     48,(30),30
>    71:  0x000058ac  main    +0008   BL      get_input,2
          0x000058b0  main    +000c   OR      0,0,0
     72:  0x000058b4  main    +0010   BL      load_month_table,2
          0x000058b8  main    +0014   OR      0,0,0
   File: DEMOCS.PUB.SYS            Procedure: main    Line: 71
  Copyright Hewlett-Packard Co. 1985.  All Rights Reserved.
  <<<< XDB Version A.02.00 MPE/XL>>>>
  Procedures: 7
  Files: 3
  >
```

LG200120_004a

Figure 3-3.
The HP Symbolic Debugger Screen (Source and Disassembly Mode)

To return to source mode, enter `ts` again.

Stepping through the Program

The debugger lets you step through a program one (or more) statements at a time. If you're in disassembly mode, you execute one or more machine instructions; if you're in source mode, you execute one or more source statements. If you're in split-screen mode, the single step mode (symbolic or assembly) is indicated on the highlighted line separating the source window from the assembly window.

Stepping lets you closely examine program execution. During stepping, you can display and alter variables or perform other tasks.

The following command executes the next six statements (or machine instructions) then pauses:

```
>s 6
```

To repeat the step command, press RETURN or type a tilde (˜) followed by RETURN.

If the program contains procedure calls and you do not want to step through the code in the procedures themselves individually, use the *S* (*Step*) command. The procedure call statements (or instructions) are treated as one step. To single step through a program and to treat procedure calls as one step, enter:

```
>S
```

Searching for a String in the Current File

This section explains how to locate certain text elements in the current source file. For example, you can search for array elements and pointers by name or you can search for arithmetic expressions. You can search forward or backward in the current file for any text string. When you reach the end of the current file, searching starts again at the beginning. Likewise, when searching backwards and you reach the beginning of the current file, searching continues at the end of the file.

The following example searches forward in the program for the string *r:= 0* and stops at the first occurrence of it.

```
>/r:= 0
```

To search backward in a program for the string `const n = 10`, enter:

```
>?const n = 10
```

String searches can be case sensitive or case insensitive. Use the *tc (toggle case)* command to control case sensitivity.

Search strings will be matched exactly (possibly disregarding case). All characters are significant, including blank spaces. If no match is found, the current viewing location does not change. Note that after locating an occurrence of the search string, the debugger may not always know what procedure the string was found in and will display `Procedure: Unknown` in the location window.

Note

To repeat a previous search command searching in the same direction, enter the *n* (*next*) command. To repeat the previous search command but search in the opposite direction, enter the *N* (*Next*) command.

Pausing during Execution

When you want to temporarily suspend the execution of the program to examine some aspect of it, such as a variable's value, set one or more *breakpoints* in the program. This must be done before starting the program, or when it is suspended by an existing breakpoint or an exception condition.

Breakpoints direct the debugger to stop execution at or immediately before executing the specified line (or instruction). When you resume execution, the program will continue until this or another breakpoint is reached. While the program is suspended, you can enter any debugger command.

Setting Breakpoints

To set a breakpoint in source mode, enter the line number before which you want execution to pause. In disassembly mode, enter an address or a label and offset to specify the location for the breakpoint. If it is not an executable statement, the debugger sets a breakpoint at the first executable statement following that line. You can set breakpoints before step and run commands or after another breakpoint occurs.

The following example sets a breakpoint before line 10:

```
>b 10
```

When a breakpoint is set, the debugger displays in the command window the procedure and line number where the breakpoint is set and the source statement located at that line. If your terminal supports windowing, the line is marked in the source window with an *asterisk* (*). From this point on, the debugger pauses each time line 10 is encountered.

To pause after a specific number of occurrences of the breakpoint, enter the *b* (*breakpoint*) command followed by a number. In the following example, a breakpoint is set at line 10. The debugger pauses every other time line 10 is encountered.

```
>b 10 \2
```

To set a breakpoint at the first executable statement in every debuggable procedure in the program, enter:

```
>bp
```

To execute a series of debugger commands before each procedure is executed, enter the *bp* (*breakpoint procedure*) command with a command list. For example, to track the value of a particular variable, the following command sets a breakpoint at the beginning of each procedure and executes three commands (*Q*, *p* and *c*) at each of these breakpoints.

```
>bp {Q; p someglobal; c}
```

In this example, the *Q* (*Quiet*) command suppresses the debugger messages that are normally displayed when a breakpoint is encountered. The *p* (*print*) command displays the current value of the global variable *someglobal*. The *c (continue)* command resumes execution of the program.

You can also set all-procedure breakpoints with the *bpt* and the *bpx* commands. The *bpt* command sets a trace breakpoint at the beginning and exit of all procedures. The *bpx* command sets a breakpoint at each procedure's exit.

For HP COBOL II programs, use the *bpg* (*breakpoint paragraph*) and the *tpg* (*trace paragraph*) commands to set all-paragraph breakpoints.

The procedure and paragraph breakpoints are set for all procedures and paragraphs. You cannot set individual procedure breakpoints in this manner. The *b* (*breakpoint*) command can be used to set individual procedure or paragraph breakpoints, which will co-exist with any all-procedure or all-paragraph breakpoint that may be set at the same location.

Resuming Execution After a Breakpoint

Once the debugger pauses for a breakpoint and you have finished entering commands at that breakpoint, enter the *c* (*continue*) command:

```
>c
```

This causes execution to continue until another breakpoint is encountered or the program terminates.

Listing Breakpoints

To list the breakpoints that are set in the program, enter the *lb* (*list breakpoints*) command as follows:

```
>lb
```

When the *lb* (*list breakpoints*) command is executed, information about each breakpoint is displayed. For example, two breakpoints are shown below. The first number on each breakpoint line is the debugger-assigned breakpoint number, which you use with other commands (such as *db* (*delete breakpoint*)). The number following *count* is the number of times the source statement will be encountered before the program pauses. The breakpoint state (active or suspended) is listed next, followed by the line at which the breakpoint is set and the source statement on that line.

```
Overall breakpoints state:  SUSPENDED
    1: count: 1 Active      sortall: 12: abc += 1;
    2: count: 5 Suspended fixit: 29: def=abc >> 4;
```

Deleting Breakpoints

To delete a breakpoint, enter the debugger-assigned number of the breakpoint (see the previous section "Displaying Breakpoints") with the *db* (*delete breakpoint*) command.

For example, to delete the breakpoint whose number is 2, enter:

```
>db 2
```

If you do not enter the breakpoint number, the breakpoint at the current line, if any, is deleted. If there is no breakpoint at the current line, the debugger displays all of the breakpoints.

To delete all breakpoints, enter:

```
>db *
```

To delete all-procedure breakpoints (only those breakpoints set by the *bp* (*breakpoint procedure*), *bpt* (*breakpoint trace*), or *bpx* (*breakpoint exit*) commands), enter the following respective commands:

```
>dp

>Dpt

>Dpx
```

To delete all-paragraph breakpoints (breakpoints set by the *bpg* (*breakpoint paragraph*) or *tpg* (*trace paragraph*) commands), use the *dpg* (*delete paragraph*) command:

```
>dpg
```

Displaying Data

Whenever program execution pauses, you can display the contents of simple variables, arrays, structures and pointers.

For HP COBOL II programs, use the *disp* (*display*) command. You can display simple items, fields, array elements, and expressions. Types of items displayed can be *edited* or *non-edited*.

The following example displays the variable X:

```
disp X
```

For HP FORTRAN 77, HP Pascal, or HP C programs, use the *p* (*print*) command. Various options and formats are available for greater control over displaying data.

The example below shows how to display the value of the variable *fob* in a form that is consistent with the language used (if the variable is an integer variable, for example, the value is expressed in decimal form):

```
>p fob
```

To display a variable or expression in a hexadecimal format, enter a print command in a form similar to this:

```
>p fob\x
```

To interpret an expression as a long integer, enter the print command in this form:

```
>p hanoi\D
```

To display the next data item using the current format (the format most recently used) and data item size, enter the print command in this form:

```
>p+
```

This interprets the next sequential data item after the one previously printed.

To display the next data item using a format different from the current one, use this form:

```
>p+ \x
```

To display the previous data item using the current format and data item size, enter the print command in this form:

```
>p-
```

To display the previous data item using a format different from the current one, use this form:

```
>p- \x
```

p+ and *p-* are best used to traverse the elements of an array.

To display the variable used with the last command, enter:

```
>p .
```

To display the contents of the location that is 30 bytes ahead of the last displayed data item in memory (HP C), enter:

```
>p *(&.+30)
```

This assumes the specified location begins a data item of the same type and size.

Modifying Data

When you need to alter the value of a variable, array item or pointer, use the *mov* (*move*) command for HP COBOL II programs. The *mov* (*move*) command requires a source and destination. The **source** can be any non-edited field, a string literal, a number, a named constant ("spaces" or "blanks"), or an expressions involving any of these data elements. The destination can be any non-edited field.

The following example sets *y* to the value of the expression *x+24*.

```
mov x+24 to y
```

For HP FORTRAN 77, HP Pascal, and HP C programs, use the *p* (*print*) command followed by an expression that contains an assignment operator. Enter the expression in the same syntax as the language in which the program is written.

This example changes the value of the variable *A1* to 30 (HP C or HP FORTRAN 77):

```
>p A1=30
```

The following example sets the variable *j* to the value of the expression *j + 17*:

```
>p j = j + 17
```

In HP Pascal, this same example is:

```
>p j := j + 17
```

Tracing Function and Procedure Calls

When a program contains several functions or procedure calls, you might need to know the sequence of calls that led to the current point of suspension. Displaying this sequence is called “viewing the stack”. To view the stack, enter the *t* (*trace*) command:

```
>t
 0 f2 (i = 3)    [t.c: 17]
 1 f1 (i = 2)    [t.c: 11]
 2 main ()    [t.c: 5]
```

The debugger lists the current (depth 0) procedure first.

Capturing and Rerunning a Debugger Session

If, before a debugging session, you think you might need to retrace your steps, you can capture the debugger commands you used during the session. You can save the debugger commands in a file and "play them back" during a subsequent session.

To write the debugger commands to a file, start the debugger using the *-r* option. The example below starts the debugger and directs it to write all commands to the file *acdebug*:

```
XDB;INFO="-r ACDEBUG TEST1P"
```

To play back the file in subsequent debugger sessions, enter this command:

```
XDB;INFO="-p ACDEBUG TEST1P"
```

This file may also be played back from inside the debugger using the < command:

```
>< ACDEBUG
```

Executing Commands At Each Source Line

When you suspect that bugs might be occurring at several places in a program, or you have a bug that is especially difficult to track down, you can direct the debugger to execute one or more commands before *every* source statement is executed. For example, you might want to track the value of one or more variables through a series of detailed calculations.

The commands that you execute are called assertions. Assertion command lists must be enclosed in braces.

The following example shows how to display the variables *payw8* and *paynet* before each source statement is executed:

```
>a {p payw8; p paynet}
```

The *if* command is very useful in assertion and breakpoint command lists. For example, if *paynet* should always be less than 23000, but its value becomes greater, the assertion:

```
>a {if (paynet >=23000) {x}}
```

will stop the program when *paynet* exceeds the legal value.

Using Macros

Macros are words that represent one or more debugger commands. You create macros by entering names for them and specifying the commands for which they stand. Macros are very useful for representing a group of commands that you execute often. You do not have to re-enter the commands; just enter the macro name for them.

The following command defines the macro *xyz*. Every time *xyz* is used, the commands `b 10`, `b 20`, `lb` and `r; info="test1p"` are executed:

```
>def xyz b 10 {}; b 20 {}; lb; r ; info="test1p"
```

The braces ({}) indicate that no command list is used with the commands. This is required if you want to have a breakpoint command followed by another command on the same line. Without the braces, the debugger would assume the rest of the line to be the breakpoint's command list.

Macro expansion can be enabled or disabled with the *tm* (*toggle macros*) command. Initially, macro expansion is disabled.

Altering the Execution Sequence

When the program is paused at a breakpoint or you are stepping through it, you can change the normal execution sequence of the program and cause it to resume at a different line. To resume execution of a program at a specific line, use the *g* (*goto*) command with the appropriate line number. The new line must be in the same procedure or paragraph as the current one.

The following example directs the debugger to change the next line to execute to be line 600:

```
>g 600
```

Use a *continue* or *step* command to begin execution at line 600.

```
>c
```

Getting Help

When you need help with the format of a debugger command or can't remember which command performs a particular function, use the *h* (*help*) command as follows:

```
>h
```

Help text is displayed one window at a time when the `more` feature is activated. You can use the terminal keys *PgUp*, *PgDn*, *Home*, *End* and the (CTRL)*arrow* keys (or the equivalent scroll keys on your terminal) to scroll the command window. The debugger displays enough lines to fill the command window then displays `--More--`at the bottom.

Use one of the following commands to continue.

(SPACE BAR)	Displays one more window-full.
(RETURN)	Displays one more line.
`q`	Quits scrolling and ignores the rest of the help information until another debugger prompt is issued.

If the `more` feature is suspended, help text is displayed in a continuous stream. To temporarily suspend scrolling, press (CNTL)S. To resume the display, press (CNTL)Q.

Note

You can activate and suspend the `more` feature with the *am* (*activate more*) and *sm* (*suspend more*) commands. For more information, see the listings for these commands in chapter 4 "HP Symbolic Debugger Commands".

4

HP Symbolic Debugger Commands

This chapter describes the commands recognized by the HP Symbolic Debugger. These commands are arranged by function in alphabetical order and can be entered in short form (abbreviated) or long form (spelled out). If you use the long form, space between command words is optional.

Entering Commands

The HP Symbolic Debugger keeps track of the current file, procedure, line and data locations of the executing program. The current file, procedure, and line are always displayed in the source and location windows, but their values do not necessarily correspond to the point at which execution is suspended. However, the debugger always knows at any point in time where to continue execution. For example, you can stop execution to view a different source file, then continue where you left off.

Most debugger commands assume that the command applies to the current location and its scope. For example, if you stop in procedure *abc* and then view procedure *def* and ask for the value of a local variable that exists in both, the debugger returns the value of that variable as it exists in *def*.

Note

def must be a caller of *abc*, or the variable must be statically declared, for its value to be meaningful.

The general format of most debugger commands is:

```
command  [ location]  [ command arguments]  [ command-list]
```

Commands are one- or two-word names or abbreviations for these names. A *location* is a particular line, procedure or file. (It can also be an address in some instances). *Command arguments* are explained in the description of each command, in this chapter and a *command list* is a sequence of commands separated by semicolons.

Using Uppercase and Lowercase

Some HP Symbolic Debugger commands are case-sensitive. The two cases are treated differently by the debugger. For example:

s or step	Lowercase "s" tells the HP Symbolic Debugger to single step to the next executable statement and step into a procedure, if necessary.
S or Step	Uppercase "S" tells the HP Symbolic Debugger single step to the next executable statement and step over a procedure call.

Abbreviating Commands

You can enter commands in their complete spelled-out form (long form) or in an abbreviated form (short form). Generally, you can abbreviate one-word commands using the first character of the word. Abbreviate two-word commands using the first character of each word in the command (do not leave a space between the two characters). If you use the long form, you can leave a space between words. For example:

$\left\{ \begin{array}{l} \texttt{w} \\ \texttt{window} \end{array} \right\} number$	Changes the size of the source window.
$\left\{ \begin{array}{l} \texttt{db} \\ \texttt{delete breakpoint} \end{array} \right\} \left[number \right]$	Deletes the selected breakpoint number.

Some debugger commands are not abbreviated by following the previous rules. Refer to the individual command syntax in this chapter to find abbreviations for these commands.

Entering Variable Names

When using HP Symbolic Debugger, use the same names for variables as are used in the source program. Global variables may be preceded by a colon to distinguish them from local variables of the same name. In the following example, the global variable *gvar* is preceded by a colon to distinguish it from a local variable *gvar*.

```
p :gvar
```

Note

Use of variable names in debugger commands is normally case insensitive; for example, *gvar* is the same variable as *GVAR*. This may be changed with the *tc* (*toggle case*) command.

In addition to program variables the debugger allows you to use **special variables**. Special variables are used like regular program variables, but are maintained by the debugger. You assign values to them using the assignment operator of the language you're using. You can only assign integer values to special variables.

Special variables have names that are prefixed by a `$`. Some special variable are predefined and have special meaning. Other special variables are user-defined, variables to which the user can assign values. Special variable names can be up to 98 characters long, but it is recommended that you limit the names of special variables to 80 characters long for display purposes. The first time you reference special variables, they are created and set to their initial values. Special variables can be used for the duration of the debugging session or you can redefine them.

For example, if you enter the following command (in HP FORTRAN 77 or HP C),

```
p $xyz = 3*4
```

the special variable **$xyz** is created and assigned the value of 12.

To view special variables (except hardware registers), use the *ls* (*list specials*) command. There are several special variables that are available; all but user-defined special variables are predefined by the debugger. The special variables are:

- $*var*
  Represents user-defined variables. They are of type long integer and do not take on the type of any expressions assigned to them.
- Hardware Registers

`$r0` ... `$r31`	General Registers
`$f0` ... `$f31`	Floating Point Registers
`$pc`	Program Counter
`$sp`	Stack Pointer
`$dp`	Data Pointer
`$arg0` ... `$arg3`	Argument Registers
`$ret0` ... `$ret1`	Return-value Registers

  Represents the HP-PA registers. Some registers have both a register number and a "register use" name. For example, $*arg3* and $*r23* refer to the same register. The *lr* (*list registers*) command provides a list of accessible registers and their contents. See the section "Data Viewing and Modification Commands" in this chapter for more information. For example:

  ```
          >p $sp\x
          $sp = 0x68023208
  ```

  For more information on the HP-PA registers, refer to the *HP Precision Architecture and Instruction Reference Manual* and the *HP Procedure Calling Conventions Manual*.
- `$result`
  References the return value from the last procedure called from the command line. $*short* and $*long* are used as other ways of viewing $*result*. Where possible, $*result* takes on the type of the procedure. $*result* is only meaningful if an integer value is returned.
- `$lang`

  Allows you to view and modify the current source language flag for expression evaluations. Valid values for $*lang* are *COBOL*, *FORTRAN*, *Pascal*, *C*, and *default*. For example, if $*lang* is set to C, the debugger expects HP C syntax, regardless of the language you are debugging.

  When $*lang* is set to "default", any language expression syntax is expected to be the same as the source language of the procedure currently being viewed.
- `$line`
  Displays the current source line number (the next statement to be executed).
- `$malloc`
  Allows you to see the amount of memory (in bytes) currently

allocated by the debugger for its own use. This does not reflect memory-use of the program being debugged.

- `$step`
  Allows you to see and change the number of machine instructions the debugger steps through while in a non-debuggable procedure, before setting an uplevel breakpoint and free-running to it. (this is where a breakpoint is set immediately after the return location in the non-debuggable procedure's caller). This situation occurs only when the program is executing in a single step or assertion mode.

Entering Expressions

An expression is a symbolic or mathematical representation. Expressions consist of variables, constants and operators, or any syntactically correct combination of these items. The HP Symbolic Debugger evaluates user expressions as if they are part of the high-level language being debugged and, therefore, uses the same operators and assignment rules as the high-level language.

See Appendices B, C, D, and E for a list of operators that you can use with each language. Note that the symbolic debugger tries as much as possible to let you write expressions with the same syntax as the current language. You can change the current language by setting the value of the special variable `$`*lang*. By default, this variable is set to the language of the program you are debugging.

The `$`*in* operator, a special unary operator, evaluates to true (1) if the operand is a debuggable procedure and if `$`*pc* (the current child process program location) is in that procedure; otherwise, `$`*in* is false (0). For example, `$in load_month_table` is true if the child process is currently suspended in *load_month_table*. Also, `$`*addr*, the unary operator for retrieving the address of a variable and `$`*sizeof*, another unary operator for retrieving the byte size of a variable, are available for all languages.

The rules in each language for character and string constants are as follows:

- For HP COBOL II, HP FORTRAN 77 and HP Pascal, string constants are represented by one or more characters, enclosed by single quotation marks (') or double quotation marks (").
- For HP C, single quotation marks enclose single characters for character constants. Double quotation marks enclose zero or more characters for string constants.
- Character and string constants can contain standard backslashed escapes as understood by the HP C compiler, including those shown in table 4-1.

Table 4-1. Escape Sequences

Character	Description
backspace	\b
form feed	\f
carriage return	\r
horizontal tab	\t
vertical tab	\v
backslash	\\
single quote	\’
double quote	\”
bit pattern	\nnn (*octal digits*)
new line	\n

- Expressions can also contain the symbolic constants listed in table 4-2.

Table 4-2. Symbolic Constants

Language	Constants
HP COBOL II	SPACES ZEROS
HP Pascal	NIL MAXINT MININT TRUE FALSE
HP FORTRAN 77	.TRUE. .FALSE.
HP C	*NONE*

If you do not have an active child process, you can only evaluate expressions containing constants.

Floating point constants must be of the form:

$$\mathit{digits}\texttt{.}\mathit{digits}\begin{bmatrix}\texttt{e}\\ \texttt{E}\\ \texttt{d}\\ \texttt{D}\\ \texttt{l}\\ \texttt{L}\\ \texttt{+}\\ \texttt{-}\end{bmatrix}\mathit{digits}$$

For example, any of the following is in the correct form:

```
1.0

3.14e8

26.62D-31
```

One or more leading digits is required to avoid confusion with . (*dot*). A decimal point and one or more following digits is required to avoid confusion for some command formats. If the exponent does not exactly fit the pattern shown, it is not taken as part of the number, but as separate tokens. The `d` and `D` exponent forms are allowed for compatibility with HP FORTRAN 77. The `l` and `L` exponents forms are allowed for compatibility with HP Pascal.

In the absence of a suffix character, the constant is assumed to be of type `double` (8 byte IEEE real).

Expressions approximately follow the *HP C* language rules of promotion. In other words, `char`, `short`, and `int` become `long` and `float` becomes `double`. If either operand is a `double`, floating math is used. If either operand is `unsigned`, unsigned math is used. Otherwise, `normal` (`integer`) math is used. Results are then cast to proper destination types for assignments.

If a floating point number is used with an operator that does not normally permit it, the number is cast to `long` and used that way. For example, the HP C binary number ~ (bit invert) applied to the constant `3.14159` is the same as `~3`.

Note that `=` means *assign* in all languages but HP Pascal and HP COBOL II; to test for equality, use `.EQ.` for HP FORTRAN 77 and `==` for HP C.

In HP Pascal, `=` is a comparison operator; use `:=` for assignments. For example, suppose you invoked the debugger on a C program, then set `$lang` to `Pascal` using this command:

```
p $lang = Pascal
```

If you want to return to HP C, you must use the `:=` operator as follows:

```
p $lang := C
```

If you invoked the debugger on an HP COBOL II program, then you would use the `move` command as follows to return to HP Pascal:

```
p move Pascal to $lang
```

You can dereference any constant, variable, or expression result using the HP C `*` operator. If the address is invalid, an error is given.

Type casting is allowed. For simple types, the syntax is identical to HP C. For example:

```
(short) size
(double *) mass_ptr
```

These casts are limited to `char`, `short`, `long`, `int`, `unsigned`, `float`, `double`, approximate combinations of these keywords, and single level pointer types. Also supported are structure and union pointer type dereferences. For example:

```
bat_ptr = &bat
(struct fob) &bat
(struct fob) bat_ptr
```

Both of these casts treat `bat` as a struct of type `fob` during printing. Structure and union pointer casts can only include the keyword *struct* or *union* and an appropriate tag. No pointers (`*`) are allowed. The argument of the cast is simply treated as an address.

Whenever an array variable is referenced without giving all its subscripts, the result is the address of the lowest element referenced. For example consider the following declared arrays:

HP FORTRAN 77	`x(5,6,7)`
HP Pascal	`x[1..5,2..6,3..7]`
HP C	`x[5][6][7]`

Referencing it simply as `x` is the same as the following:

HP FORTRAN 77	`x(1,1,1)`
HP Pascal	`x[1,2,3]`
HP C	`x`

If a not-fully-qualified array reference appears on the left side of an assignment, the value of the right-hand expression is stored into the element at the address specified.

String constants are stored in a buffer in the *xdbend.lib.sys* file which you link with your program. The debugger starts storing strings at the beginning of this buffer, and moves along as more assignments are made. If the debugger reached the end of the buffer, it goes back and reuses it from the beginning. This does not usually cause any problems. However, if you use very long strings, or if you assign a string constant to a global pointer, problems could arise.

Entering Procedure Calls

You can include calls to procedures in expressions. You can call any executable procedure from the command line whether or not it was compiled with debugger information. You can use the *lp* (*list procedures*) command to list procedures in an executable program file. The following command evaluates an expression that calls the procedure *ref* and uses its return value:

```
p $xyz = $abc*(3 + ref (ghi - 1, jkl, "Hi Folks"))
```

An argument list must follow each procedure call, even if it is empty. When a procedure is called, the following might occur:

- The HP Symbolic Debugger has one active command line at a time. During command line procedure calls, breakpoints reached during program execution are treated as usual (by suspending execution as specified). If execution stops in a called procedure, the remainder of the old command line is ignored and you are informed of this.
- If you try to call a procedure when the child process is not active, then a child process is started by the debugger. This process is similar to using the single step command after starting the debugger.

Window Mode Commands

Window mode commands let you control what is displayed on the screen. The window mode commands are:

- fr (floating point registers)
- gr (general registers)
- sr (special registers)
- td (toggle disassembly)
- ts (toggle screen)
- u (update)
- U (Update)
- w (window)

The source window displays source lines in a program. In disassembly mode, the top five lines of the screen show the floating point, general or special registers (the register window) followed by assembly language instructions (the assembly window). In split-screen mode, the top part of the screen displays source code followed by the corresponding assembly language instructions.

fr (floating point registers)

$$\left\{\begin{array}{l}\texttt{fr}\\ \texttt{floating point registers}\end{array}\right\}$$

Displays the HP-PA floating point registers in the register window when the debugger is in disassembly mode. Each register appears as a two-word pair (two sets of eight hexadecimal digits). When the value of a register changes, that register is highlighted until after the next command. Use the special variables $*f0* through $*f15* to modify these registers.

Caution

Modification of floating point registers is not recommended during normal debugger usage.

For more information about the HP-PA floating point registers, see appendix F “Registers Displayed by HP Symbolic Debugger in Disassembly Mode” or refer to the *HP Precision Architecture and Instruction Reference Manual*.

gr (general registers)

$$\left\{\begin{array}{l}\texttt{gr}\\ \texttt{general registers}\end{array}\right\}$$

Displays the HP-PA general registers in the register window when the debugger is in disassembly mode. When the value of a register changes, that register is highlighted until after the next command. Use the debugger's special variables $*r0* through $*r31* (or equivalent usage names such as $*arg3*) to modify these registers.

When displaying the general registers or the floating point registers, the line dividing the registers from the assembly code displays the current *space* number, pc offset, privilege level, *sar* and *psw*. The *psw* is displayed as a string of letters, with each letter representing one flag bit in the *psw*. For example, `b` stands for `branch taken trap` and `n` stands for `nullify`. A lower case letter indicates that the bit is *OFF* while upper case indicates it is *ON*. You cannot modify the *sar* or *psw* registers. For more information about the HP-PA general registers, see appendix F "Registers Displayed by HP Symbolic Debugger in Disassembly Mode" or refer to the *HP Precision Architecture and Instruction Reference Manual*.

sr (special registers)

$$\left\{\begin{array}{l}\texttt{sr}\\ \texttt{special registers}\end{array}\right\}$$

Displays the special registers (space and control) when the debugger is in disassembly mode. When the value of a special register changes, that register is highlighted until after the next command. You cannot modify the special registers. For more information about the HP-PA special registers, see appendix F "Registers Displayed by HP Symbolic Debugger in Disassembly Mode" or refer to the *HP Precision Architecture and Instruction Reference Manual*.

td (toggle disassembly)

$$\left\{\begin{array}{l}\texttt{td}\\ \texttt{toggle disassembly}\end{array}\right\}$$

Toggles the source window between disassembly mode and source mode. When in disassembly mode, this command displays the assembly language instructions that correspond to the source code as well as one of the three sets of registers (floating point, general or special).

When in disassembly mode, the single step command steps one machine instruction at a time (rather than one source statement at a time). The assembly language display of each instruction consists of: the source line number, the address in hexadecimal, the address in the form of the nearest label plus the offset and the actual symbolic assembly instruction and operands.

ts (toggle screen)

$\left\{\begin{array}{l}\texttt{ts}\\ \texttt{toggle screen}\end{array}\right\}$

Toggles the source window between all source or all assembly and split-screen mode. In split-screen mode, the source window displays both source code and corresponding assembly instructions. Single stepping occurs at either the source statement or the assembly instruction level, depending on the part of the split-screen in which you are single stepping. The stepping mode is displayed in the line separating the source and assembly windows.

u (update)

$\left\{\begin{array}{l}\texttt{u}\\ \texttt{update}\end{array}\right\}$

Updates the source and location windows to show the current location of the user program. This command is useful in an assertion. For example, this command:

```
a {u}
```

will continuously update the screen to show the execution of the program as it runs.

U (Update)

$\left\{\begin{array}{l}\texttt{U}\\ \texttt{Update}\end{array}\right\}$

Clears the screen of data and redraws the screen. Use this command if the screen gets corrupted by a system-wide announcement that overwrites your session.

w (window)

$\left\{\begin{array}{l}\texttt{w}\\ \texttt{window}\end{array}\right\}$ *number*

If your terminal supports windowing, this command changes the size of the source window to the number of lines that you specify. Enter a number from 1 to 21 (the default is 15). Changing the size of the source window also changes the size of the command window.

If your terminal does not support windowing, this command prints the specified number of lines surrounding the current line. If no number is specified, the last number used with the *w* (*window*) command is used again. You can press RETURN to repeat this command. The next specified number of lines will be displayed.

File Viewing Commands

The file viewing commands let you view program source code. The file viewing commands are:

- +
- -
- /
- ?
- D (Directory)
- ld (list directories)
- lf (list files)
- L (Location)
- n (next)
- N (Next)
- v (view)
- V (View)
- va (view address)

+

`+ [number]`

Moves forward in the current file the specified number of lines (or the specified number of instructions in disassembly mode). If you do not enter a number, the next line (or instruction) becomes the current line (or instruction).

You can press a RETURN to repeat this command. If your terminal supports windowing, a new group of lines are displayed. If it does not support windowing, only the new current line and its line number are displayed.

-

`- [number]`

Moves the specified number of lines (or the specified number of instructions in disassembly mode) backward in the current file and updates the windows. The default is one line (or instruction) before the current line (or instruction).

You can press RETURN to repeat this command. If your terminal supports windowing, a new group of lines (or instructions) are displayed. If it does not support windowing, only the new current line and its line number are displayed.

/

`/ [string]`

Searches forward in the file for the specified string. Searches wrap around the end of the file. If you do not enter a string, the last one that you entered is used again. The string must be literal; wild cards are not supported.

You can select case sensitivity for string searches with the *tc* (*toggle case*) command. Initially, searches are case insensitive.

?

$? \; [\textit{string}]$

Searches backward in the current file for a specific pattern. Searches wrap around the beginning of the file. If you do not enter a string, the last search string is used again. The string must be literal; wild cards are not supported.

You can select case sensitivity for string searches with the *tc* (*toggle case*) command. Initially, searches are case insensitive.

D (Directory)

$\left\{\begin{array}{l}\texttt{D}\\\texttt{Directory}\end{array}\right\}\texttt{"}\textit{dir}\texttt{"}$

Adds the directory that you specify to the list of directory search paths for source files. You can add more that one directory, but only one can be added at a time. Directories are searched in the order that they are added.

ld (list directories)

$\left\{\begin{array}{l}\texttt{ld}\\\texttt{list directories}\end{array}\right\}$

Lists all the alternate directories that are searched when the debugger tries to locate the source files.

lf (list files)

$\left\{\begin{array}{l}\texttt{lf}\\\texttt{list files}\end{array}\right\}[\textit{string}]$

Lists all source files containing executable statements that were compiled to build the executable file. Code address ranges are shown for each file. Only files containing executable code are shown. If a string is specified, only those files beginning with this string are listed.

This command also lists any include files containing executable code with their code addresses. A file can appear several times if it contains include files. An example of the output is:

```
0:  STDIO.PUB.C                0x00001834 to 0x00002524
1:  TREE1.SRC.PROJECT          0x00002530 to 0x00003210
2:  TREEGLOBS.PUB.PROJECT      0x00003344 to 0x00004002
3:  TREE2.NEWSRC.PROJECT       0x00004040 to 0x00006832
```

L (Location)

$$\left\{\begin{array}{l}\texttt{L}\\\texttt{Location}\end{array}\right\}$$

Displays in the command window the current file, procedure, line number and the source text for the current point of execution. This command allows you to determine where you are in the program and is useful when included in an assertion or breakpoint command list. For example:

```
>L
doproc.c: eval_q: 8: if (qp != NULL) {
```

You cannot press RETURN to repeat this command.

n (next)

$$\left\{\begin{array}{l}\texttt{n}\\\texttt{next}\end{array}\right\}$$

Repeats the previous search (/ or ?) command.

N (Next)

$$\left\{\begin{array}{l}\texttt{N}\\\texttt{Next}\end{array}\right\}$$

Repeats the previous search (/ or ?) command, searching in the opposite direction.

v (view)

$\left\{\begin{array}{l}\texttt{v}\\\texttt{view}\end{array}\right\}[\,\textit{location}\,]$

Displays one source window forward from the current source window. One line from the previous window is preserved for context. If your terminal does not support windowing, only the new source line is displayed.

A location can be a particular line, procedure, or any text file, whether used in the program or not. For COBOL, a location can also be a paragraph or section. Using the *location* option causes the specified location to become the current location, and the source at the specified location is then displayed in the source window. The source location window is adjusted accordingly.

If a procedure (`proc`) name is specified for the location, the procedure's first executable line becomes the current line.

You can press RETURN to repeat this command. If a location was given, subsequent RETURN's move forward from that point.

Note

Filenames entered with the *v* (*view*) command cannot be qualified by a path name. This means the debugger must be able to determine where the file is either by using the *-d* option when running the debugger or by using the *D* (*Directory*) command. Note that the debugger uses the same pathnames for finding source as were used during compilation.

V (View)

$\left\{\begin{array}{l}\texttt{V}\\\texttt{View}\end{array}\right\}[\,\textit{depth}\,]$

Displays the text for the procedure at the depth on the program stack that you specify. If you do not enter a depth, the current active procedure is used. This command is normally used to view the current point of suspension when the current viewing location is elsewhere in the program.

If your terminal supports windowing, the new lines are displayed in the window. Pressing RETURN lets you view successive windows. If your terminal does not support windowing, the current line (including its line number and description) is displayed. Pressing RETURN lets you view the next line in sequence.

va (view address)

$$\left\{\begin{array}{l}\texttt{va}\\\texttt{view address}\end{array}\right\}\mathit{address}$$

Displays in the source window assembly code at the specified address. A specified address can be an absolute address or symbolic code label with an optional offset (for example, _start + 0x20). This command is used in disassembly mode only.

Data Viewing and Modification Commands

Data viewing and modification commands allow you to view program data in a variety of formats and change the values of variables. The data viewing and modification commands are:

- disp (display)
- l (list)
- lc (list common)
- lg (list globals)
- ll (list labels)
- lm (list macros)
- lp (list procedures)
- lr (list registers)
- ls (list specials)
- mov (move)
- p (print)

disp (display)

$$\left\{\begin{array}{l}\texttt{disp}\\ \texttt{display}\end{array}\right\}\textit{expression}$$

Used only with HP COBOL II programs to print COBOL variables or expressions. Simple items, fields, array elements, and expressions can be displayed. Items displayed can be of type "edited" or "non-edited". For example:

```
>disp z of y of x of w(3,7,11)
0x40003028  345.67
```

This command is equivalent to the *p* (*print*) command for other source languages.

l (list)

$\left\{\begin{array}{l}\texttt{l}\\ \texttt{list}\end{array}\right\}[\,proc[\,:depth\,]\,]$

Lists all parameters and local variables of the current procedure. You can optionally specify any active procedure and its depth on the stack. If a procedure name is given without a depth, then the most recent invocation of that procedure is used. If an invocation of that procedure other than the most recent is desired, then a depth must be specified. The following illustrates the use of this command.

If the current stack trace (generated with the command `t 5`) is:

```
0  groucho( )       [marx.c:23]
1  harpo( )         [marx.c:70]
2  chico( )         [marx.c:55]
3  harpo( )         [marx.c:73]
4  main( )          [marx.c:16]
```

and *groucho* is the procedure currently viewed (where execution is currently suspended), then:

`l`	Lists the local variables and parameters of *groucho*.
`l harpo`	Lists the local variables and parameters of *harpo* at level 1 on the stack.
`l harpo:3`	Lists the local variables and parameters of *harpo* at level 3 on the stack.

The \n (normal) format is used to display the procedures, parameters, and local data except for arrays and pointers, which are displayed as addresses.

lc (list common)

$\left\{\begin{array}{l}\texttt{lc}\\\texttt{list common}\end{array}\right\}[\textit{string}]$

Used when debugging an HP FORTRAN 77 program, this command displays HP FORTRAN 77 common blocks and their associated variables. If a string is specified, only those common blocks whose names begin with that string are printed; otherwise, all common blocks within the current subroutine or function are printed.

Sample output is:

```
>lc
COMMON  /COM1/
 BR4      = 0
 INT1     = 0
 BR8      = 0
 BI4      = -2097152000
 BI2      = -32000
 BCX8     = 0
 BC1      = '\000'
 BL4      = .FALSE.
```

lg (list globals)

$\left\{\begin{array}{l}\texttt{lg}\\\texttt{list globals}\end{array}\right\}[\textit{string}]$

Lists all global variables and their values. If a string is specified, only those global variables whose names begin with this string are listed.

ll (list labels)

$\left\{\begin{array}{l}\texttt{ll}\\\texttt{list labels}\end{array}\right\}[\textit{string}]$

Lists all labels and program entry points known to the linker. If a string is specified, only those symbolic addresses with this prefix are used.

lm (list macros)

$\left\{\begin{array}{l}\texttt{lm}\\\texttt{list macros}\end{array}\right\}[\textit{string}]$

Displays all user-defined macros and their definitions. If a string is specified, only those macros whose names begin with this string are listed.

Sample output is:

```
>lm
pheadtuti ==> p flavor:list->head.tutifruti
unS ==> bu\t {}; c
Overall macros state:  ACTIVE
```

lp (list procedures)

$\left\{\begin{array}{l}\texttt{lp}\\\texttt{list procedures}\end{array}\right\}[\textit{string}]$

Lists all procedure names and their aliases as well as their locations in memory. If a string is specified, only those procedures whose names begin with this string are listed.

Sample output is:

```
0:  main          0x00001218 to 0x000013c0
0:  _MAIN_
1:  proc1         0x000013c8 to 0x00001438
2:  proc2         0x00001440 to 0x0000148c
3:  _end_         0x000047e8 to 0x000047fc
```

Note

The following notes apply to when using the *lp* (*list procedures* command):

- Only subprograms are shown when using this command with HP COBOL II subprograms.
- The procedure name *main* is used as the alias name for the main program in all supported languages. Do not use it for any debuggable procedures.
- For code that is not debuggable or does not have a corresponding source file, the debugger displays *unknown* for the unknown file and procedure names. The debugger cannot show code locations or interpret parameter lists and so on. However, procedure names are provided for most procedures, even if not debuggable.

lr (list registers)

$\left\{\begin{array}{l}\texttt{lr}\\\texttt{list registers}\end{array}\right\}[\textit{string}]$

Lists all registers and their contents. This command displays all general and floating point registers, as well as the program counter, stack pointer registers, and other registers. If a string is specified, only those registers beginning with this string are listed (the $ is significant). All register values are printed in hexadecimal.

ls (list specials)

$\left\{\begin{array}{l}\texttt{ls}\\ \texttt{list specials}\end{array}\right\}[\,\textit{string}\,]$

Lists all special variables and their values. Registers are not listed. If a string is specified, only those special variables whose names begin with this string are listed.

Sample output is:

```
$lang      = COBOL (default)
$line      = 49
$signal    = 0
$malloc    = 43008
$step      = 100
$long      = 0
$short     = 0
$result    = 0
```

$result is normally interpreted to be the same type as the last procedure call (if the call returns a structured type, *$result* defaults to integer). Note that there are two alternate ways of looking at *$result*, as a 32 bit integer (*$long*) or as a 16 bit integer (*$short*).

You can also list special variables defined by usage. For example:

```
p $var = 10
```

defines the variable *$var* to be equal to 10. The *ls* (*list specials*) command will also display *$var* and its current value.

mov (move)

$\left\{\begin{array}{l}\texttt{mov}\\ \texttt{move}\end{array}\right\}\textit{expr1}\ \texttt{to}\ \textit{expr2}$

Used only with HP COBOL II programs to modify variables. The first expression is the *source*; the second is the *destination*. The source and destination cannot be an edited field. The source can be any non-edited COBOL field, a string literal, a number, or a named constant (such as SPACES or BLANKS). The destination can be any non-edited COBOL field. For example:

```
>mov zeros to n-comp-03-u
N-COMP-03-U = .000
```

This command is equivalent to *print expr2=expr1* in other source languages.

p (print)

$$\left\{\begin{array}{l}\texttt{p}\\ \texttt{print}\end{array}\right\}\left\{\begin{array}{l} expr\ [\,?format\,] \\ \begin{bmatrix} + \\ - \end{bmatrix} [\,[\,\backslash\,]format\,] \end{array}\right\}$$

Displays and optionally modifies program data. You can choose to display data in one of the formats shown in tables 4-3 and 4-4. The *p* (*print*) command is also used to evaluate arbitrary expressions involving constants and/or program data.

A **format** has the syntax:

[*count*] { *formchar* } [*size*]

Formchar, which is required, specifies the actual format in which you choose to display the data. *Count* is the number of times to apply the format. *Size* is the number of bytes that are formatted for each data item, and overrides the default size for the given format. The count must be a decimal, octal, or hexadecimal number. The size must be a decimal number or the letters b, s, and l which are predefined sizes of 1 byte (8 bits), 2 bytes (short), and 4 bytes (long) respectively. For example:

```
>p abc\4x2
```

prints four two-byte numbers in hexadecimal starting at the address designated by the variable *abc*. If *abc* is an array, you need to specify a subscript if you want to see the contents of consecutive array elements. For example:

```
>p abc[5]\4n
```

will display four elements of array *abc* starting with element 5.

Table 4-3 lists the possible data formats and corresponding *formchars*. Note that there is usually a difference between a lowercase and uppercase character.

For example, the *d* and *D* formats print in short and long decimal:

```
d          Displays 16 bits
D          Displays 32 bits
```

Short and long form apply only to the following formats:

Short	Long
d	D
e	E
f	F
g	G
o	O
u	U
x	X

Many of the the data formats have a default size if the size is not given. For example, *X* has a default size of four bytes. There are also some shorthand notations for *size*. These shorthand notations are shown in table 4-4. Shorthand notations can be appended to *formchar* instead of a numeric size. For example:

```
xb
```

prints one byte in hexadecimal.

There is also a default for the format, if the format is not specified. For example: *D* is the default for a long integer variable or field, *X* is the default for a pointer or array variable or field, and *S* is the default for a structure variable. The *n* format specifies the default. In general, if the expression describes a named data object, the debugger will display its value in a manner consistent with the object's declared type, even if it is a structured type. if the debugger cannot determine the type of an expression or data object, *X* is used.

The following example prints a dynamically allocated C structure that is local to procedure *flavor*.

```
>p *flavor:list->head
0x68023004 struct {
    chocolate = 1597845365;
    tutifruti = 2.21414e-10;
}
```

Table 4-3. Data Viewing Formats

Formchar	Description
`a`	Prints a string using the expression as the address of the first byte.
`(b|B)`	Prints a byte in decimal.
`(c|C)`	Prints a character.
`(d|D)`	Prints in decimal as an integer or long integer, respectively.
`(e|E)`	Prints in e floating point notation as a float or double, respectively. (4 bytes, 8 bytes)
`(f|F)`	Prints in f floating point notation as a float or double, respectively.
`(g|G)`	Prints in g floating point notation as a float or double, respectively.
`n`	Prints in normal (default) format, based on the type. (if known)
`(o|O)`	Prints the expression (*expr*) in octal as an integer or long integer, respectively.
`p`	Prints the name of the procedure containing the given address. (*expr*)
`s`	Prints a string using an expression as the address of a pointer to the first byte. In HP C, this is the same as specifying **expr\a*.
`S`	Prints a formatted dump of structures, fields and their values. The expression (*expr*) must be the address of a structure, not the address of a pointer to a structure.
`t`	Shows the type of the expression (*expr*), usually a variable or procedure name.
`(u|U)`	Prints the expression (*expr*) in unsigned decimal as an integer or long integer. If the quantity is known to be a full word, *u* gives the same result as *U*.
`(x|X)`	Prints in short and long hexadecimal, respectively. If the quantity is known to be a full word, *x* gives the same result as *X*.

Table 4-4. Shorthand Notation for Size

Mnemonic	Actual Size
`b`	1 byte (8 bits)
`s`	2 bytes (16 bits)
`l`	4 bytes (32 bits)

Use the *\format* option to display the value of the expression in a specific format. For example:

```
>print abc\x
```

prints the contents of *abc* in hexadecimal. If a format is not given, the expression is displayed in a format consistent with the type of the expression. For example:

```
>print (abc*3/25)+2
```

prints the results of evaluating the given expression using the current value of *abc* in decimal. Use the *?format* option to print the address of the evaluated expression in the selected format. For example:

```
>print abc?o
```

prints the address of *abc* in octal. If the expression is not a named data item, *?* is equivalent to \.

Display absolute addresses with the *p* (*print*) command when you are debugging a program with no debugger information. For example:

```
>p *0xC0000348
```

or

```
>p *($sp-36)\x)
```

p+ prints the next element. Based on the size of the last item displayed, *p+* increments the current data address by the size of the previous format and then displays the contents of memory starting at the new address, using the format if it is supplied, or the previous format, if not supplied. This command is useful for displaying successive elements of an array. The initial *p* (*print*) command can determine the array's format by its type.

p- prints the previous element. Based on the size of the last item displayed, *p-* decrements the current data address by the size of the previous format and then displays the contents of memory starting at the new address, using the format if it is supplied, or the previous format, if not supplied.

The *p* (*print*) command is also used to modify the value of a variable. Modification of variables is done by using the assignment operator in the expression (= in HP C, HP FORTRAN 77, and HP COBOL II, or := in HP Pascal). For example:

```
>p fob=7
```

In the case of an assignment, the debugger will also show the name of the variable being modified or the address used if the expression is not a simple name.

Here are some symbol table dependencies:

1. When you try to display a variable which is an HP FORTRAN 77 format label, an HP Pascal file-of-text, or an HP Pascal set, with no display format or with normal format (\\$n$), the value is shown as {*format-label*}, {*file-of-text*}, or {*set*}, respectively. You can use other formats, such as \\$x$, to display the contents of such variables.
2. When a compiler does not know array dimensions, such as for some HP FORTRAN 77 and HP C array parameters, it uses `0:MAXINT` or `1:MAXINT` as appropriate. The \\t format shows such cases with `[]` (no bounds specified), and subscripts from 0 (or 1) to MAXINT are allowed in expressions.
3. Even though the symbol table supports C structure, union, and enumeration tags, C typedefs, and Pascal types, the debugger does not know how to search for them, even for the \\t format. They are "invisible".
4. Some variables are indirect, so a child process must exist in order for the debugger to know their addresses. When there is no child process, the **address** of any such variable is shown as `0xfffffffe`.

Stack Viewing Commands

Stack viewing commands trace the stack of a program. The stack viewing commands are:

- t (trace)
- T (Trace)

Figure 4-1 illustrates the stack depth of a program and shows that A called B, B called C, C called D, D called E, E called F, and program execution is currently suspended in F. The procedure at which the program is currently stopped is always at depth zero.

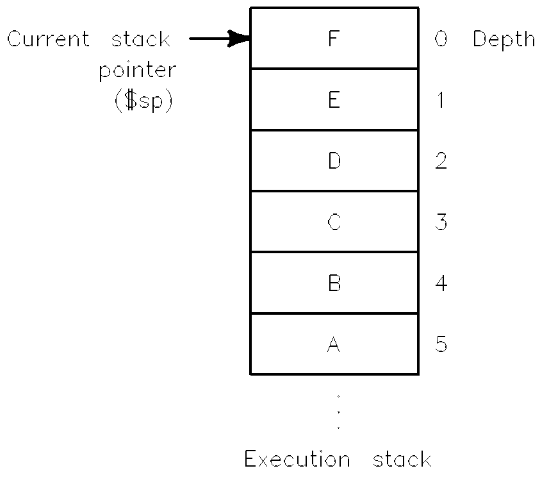


Figure 4-1. Stack Depth

t (trace)

$\left\{\begin{array}{l}\texttt{t}\\ \texttt{trace}\end{array}\right\}[\,depth\,]$

Prints a stack trace. You can optionally specify a *depth*. The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed. For each procedure in the stack trace, the following is displayed:

- Stack depth
- Name of procedure at that depth
- Name of procedure parameters and their values (printed in *normal* (\n) format)
- Source file and line number where it is suspended (depth 0) or where a call to the next procedure (at the next lowest depth) occurred.

The following example is a NON-COBOL trace.

```
>t
 0 icecream (i = 7)    [ice.c: 8]
 1 flavor (year = 1988)    [flavors.c: 19]
 2 main ()    [main.c: 59]
```

The following example is an HP COBOL II trace. Note that this trace shows both stacked subprograms as well as paragraphs.

```
>t 3
 0 PROGRAM-9 ()    [LCBTST21.V.JOE: 74]
    0 Paragraph PARA-9x2
    1 Paragraph PARA-2y
 1 PROGRAM-8 ()    [LCBTST21.V.JOE: 48]
    0 Paragraph PARA-1C
 2 PROGRAM-7 ()    [LCBTST21.V.JOE: 43]
    0 Paragraph EVAL-CK
```

T (Trace)

$\left\{\begin{array}{l}\texttt{T} \\ \texttt{Trace}\end{array}\right\}$ [*depth*]

Prints a stack trace. You can optionally specify a *depth*. The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed. For each procedure in the stack trace, the following is displayed:

- Stack depth
- Name of procedure at that depth
- Name of procedure parameters and their values (printed in *normal* (\n) format)
- All local variables and their values (printed in *normal* (\n) format)
- Source file and line number where execution is suspended (depth 0) or where a call to the next procedure (at the next lowest depth) occurred.

All arrays, structures, and pointers are shown as addresses. Only the first word of a structure is shown.

The following example is a NON-COBOL Trace.

```
>T
 0 icecream (i = 7)    [ice.c: 8]
 c         =   00000000
 1 flavor (year = 1988)    [flavors.c: 19]
 harpo    = 1995
 list     = 0x680235bc
 2 main ()    [main.c: 59]
 i        = 3
 j        = 2987
 k        = 1988
 icecream = 0x00000000
 buff     = 0x6802377e
```

When displaying a Trace of an HP COBOL II program, paragraph "stacking" is shown within the subprograms called (if any). Variables local to a given subprogram are shown. For example:

```
>T
 0 PROGRAM-9 ()    [LCBTST21.V.JOE: 74]
    0 Paragraph PARA-9x2
    1 Paragraph PARA-2y
 TALLY     = 0
 Y         = 427

 1 PROGRAM-8 ()    [LCBTST21.V.JOE: 48]
    0 Paragraph PARA-1C

 2 PROGRAM-7 ()    [LCBTST21.V.JOE: 43]
    0 Paragraph EVAL-CK
 CK-LBL    = 6536-A
```

Status Viewing Command

The status viewing commands display the state of the debugger and the program being debugged. Various *list* commands can be used. Refer to the section on Data Viewing and Modification for further information about *list* commands. The other major status viewing command is:

- Inquire

I (Inquire)

$$\left\{\begin{array}{l}\texttt{I}\\ \texttt{Inquire}\end{array}\right\}$$

Prints the current status of the debugger. The output contains information such as the version number of the debugger, program name, number of source files and procedures, process ID of the child process, number of breakpoints, record and playback information and so on. A sample output is displayed:

```
Version .......... HP31508 A.02.03 HP SYMBOLIC DEBUGGER (XDB)
Program .......... "tree"
Core File ........ None
Procedures ....... 10
Child process .... None
Breakpoints ...... 4 (Active)
Assertions ....... 3 (Suspended)
Macros ........... 9 (Active)
Recording ........ Suspended
Record file ...... None
Record-all ....... Active
Record-all file .. mysession
Playback file .... None
Searches ......... NOT case sensitive
Address format ... "%#10.81x"
Bytes malloc'd ... 7168
Run arguments .... ""
```

Job Control Commands

The job control commands let you control execution of the program. The parent (HP Symbolic Debugger) and child (*objectfile*) processes take turns running. The debugger is only active while the child process is stopped, due to encountering a signal or a breakpoint, or by terminating.

The job control commands are:

- c (continue)
- C (Continue)
- k (kill)
- r (run)
- R (Run)
- s (step)
- S (Step)

c (continue)

$\left\{\begin{array}{l}\texttt{c}\\ \texttt{continue}\end{array}\right\}[\textit{location}]$

Resumes execution after a breakpoint has been encountered, ignoring the signal, if any. If a location is specified, a temporary breakpoint is set at that location. See "Breakpoint Commands" in this chapter for more information.

C (Continue)

$\left\{\begin{array}{l}\texttt{C}\\ \texttt{Continue}\end{array}\right\}[\textit{location}]$

Resumes execution after a breakpoint has been encountered, allowing the signal, if any, to be received by the child process. If a location is specified, a temporary breakpoint is set at that location. See "Breakpoint Commands" in this chapter for more information.

Note

Since signals are currently unsupported on the MPE/iX operating system, this command will operate exactly as the *c* (*continue*) command.

k (kill)

$\left\{\begin{array}{l}\texttt{k}\\ \texttt{kill}\end{array}\right\}$

Terminates the current child process, if any. You are asked to confirm this command; this guards against accidental termination of the child process.

r (run)

$\left\{\begin{array}{l}\texttt{r}\\\texttt{run}\end{array}\right\}$ [`;info='`*info-string*`'`] [`;parm='`*number*`'`]

Lets you run a program as a new child process with an optional *info string* and *run parm*. If a child process already exists, the debugger asks if you want to terminate the child process first.

If you do not enter an *info string*, the debugger uses those supplied with the last *r* (*run*) command (if any). *info string* can contain a "<" and/or a ">" for redirecting standard input and standard output ($*STDIN*) and ($*STDOUT*).

R (Run)

$\left\{\begin{array}{l}\texttt{R}\\\texttt{Run}\end{array}\right\}$

Lets you run a program as a new child process with no argument list. If a child process already exists, the debugger asks if you want to terminate the child process first. Use this command to explicitly indicate no arguments.

s (step)

$\left\{\begin{array}{l}\texttt{s}\\\texttt{step}\end{array}\right\}$ [*number*]

Single steps through a program, executing one source statement or machine instruction at a time before pausing and prompting for another command. In source mode, one source statement is executed (or one step of a multiple step statement in HP Pascal or HP C); in disassembly mode, one machine instruction is executed (several machine instructions might be equivalent to one source statement). If a procedure call is encountered, the procedure is single stepped in the same manner (stepped "into").

Note

One *s* (*step*) is required to go from the calling statement to the first statement of the called procedure.

To execute more than one statement or instruction, enter that number as the *number* parameter. The debugger executes this number of statements or instructions before stopping unless it encounters a breakpoint.

Note

Single stepping through a procedure for which there is no debugger information (for example, printf) can be slow. You might prefer to use the *c* (*continue*) or *S* (*Step*) command instead.

If you accidentally step down into a procedure you don't care about, use the *bu* command to set a temporary "uplevel" breakpoint, and then continue using a *continue* command.

You can press RETURN to repeat this command.

S (Step)

$\left\{\begin{array}{l}\texttt{S}\\\texttt{Step}\end{array}\right\}[\,\textit{number}\,]$

Single steps through a program. In source mode, one source statement (or one step of a multiple step statement in HP Pascal or HP C) is executed; in disassembly mode, one machine instruction is executed (several machine instructions might be equivalent to one source statement). If a procedure call is encountered, it is not "stepped into". Instead, execution steps to the statement following the call. The procedure call is treated as a single statement. If a breakpoint is encountered in the procedure or any that is called, its *commands* are executed.

Note

Using a *continue* command in a breakpoint command list within a procedure will cause the program to keep executing through the procedure! If the breakpoint does not explicitly continue, the current act of stepping "over" the procedure ceases. The command list

```
>bu\t {}; c
```

continues back to the calling statement, effectively completing the *S* (*Step*) command.

The *S* (*Step*) command does not step "over" *PERFORM* statements in HP COBOL II programs, only *CALL* statements.

To execute more than one statement or instruction, enter that number as the *number* parameter. The debugger executes this number of statements or instructions unless it encounters a breakpoint.

You can press RETURN to repeat this command as a single step.

Breakpoint Commands

A **breakpoint**, when encountered, suspends the execution of the program at a particular location. HP Symbolic Debugger provides a number of commands for setting, deleting, and managing breakpoints. The breakpoint commands are:

- Overall
  - lb (list breakpoints)
  - tb (toggle breakpoints)
- Creation
  - b (breakpoint)
  - ba (breakpoint address)
  - bb (breakpoint beginning)
  - bt (breakpoint trace)
  - bu (breakpoint uplevel)
  - bx (breakpoint exit)
- Status
  - ab (activate breakpoint)
  - bc (breakpoint count)
  - db (delete breakpoint)
  - sb (suspend breakpoint)
- All-Procedures
  - bp (breakpoint procedure)
  - bpt
  - bpx
  - dp (delete procedure)
  - Dpt
  - Dpx
- Global
  - abc
  - dbc
- All-Paragraph (HP COBOL II only)
  - bpg (breakpoint paragraph)
  - dpg (delete paragraph)
  - tpg (trace paragraph)
- Auxiliary
  - "*any string*"
  - i (if)
  - Q (Quiet)

Once a breakpoint has been encountered during program execution, you can examine the program state, unless the breakpoint command list includes a command that causes the child process to continue or terminate. Examples of these commands are the *c* (*continue*), *r* (*run*), *k* (*kill*) and *q* (*quit*) commands. Individual breakpoints are identified by a unique number, which is assigned by the debugger.

Breakpoints can be activated or deactivated (suspended) individually. When a breakpoint is suspended, information for that breakpoint is retained, but it will not affect program execution. There is also an overall breakpoint mode for breakpoint activation and suspension, which is independent of the state of any individual breakpoint. Any given breakpoint will affect program execution only if it is individually activated and the overall mode is activated. Any active breakpoint whose location is visible in the source window will be marked with an asterisk (*) in the leftmost screen column. Note that only those breakpoints that are associated with a line number are marked in the source window in source mode. In disassembly mode, all breakpoints are displayed whether associated with a line or machine instruction.

Three parameters are associated with breakpoint commands, *location*, *count* and *command list*. These parameters are described below:

Location

You can set a breakpoint at the current **location** (where the prompt (>) appears in the source window) or at any other executable statement or instruction. You can specify the location of the breakpoint in a variety of ways:

- line number
- procedure name
- label
- symbolic address (with or without offset)
- absolute (numeric) address

Each of these ways of specifying a location is simply an alternate way to specify the breakpoint address. The breakpoint is encountered whenever the *location* is about to be executed, regardless of the path taken to get there. Only one breakpoint at a time of a particular type may be set at a given *location*. Setting a new breakpoint at the same location replaces the existing one.

Count

The count is the number of times a breakpoint statement or instruction is executed before the program is stopped. If the count is positive, the breakpoint is permanent. If the count is negative, the breakpoint is temporary and is cleared when the program stops there. When you enter a new breakpoint without a count, the breakpoint is permanent and a count of 1 is used. To change the count of an existing breakpoint, use the *bc* (*breakpoint count*) command.

An alternate way to enter a breakpoint is to enter *t* for a temporary breakpoint and *p* for a permanent breakpoint. Entering a *t* or *p* by itself sets the count to -1 or 1, respectively. You can precede a *t* or *p* with a positive integer for a count of more than one.

Command List	A **command list** is one or more commands that are executed when its associated breakpoint occurs. Separate commands in a command list by semicolons. Use braces {} to separate the breakpoint command list from other debugger commands on the same line.

Caution

Only one active *command line* can exist at one time. A *command line* is either the sequence of commands you enter at the debugger prompt or the command list associated with a breakpoint or assertion. If a new command line is encountered before all commands in the previous command line are executed, those remaining commands are discarded. For example, suppose you set a breakpoint in a function called *func1* which has the following command list:

```
{Q;p "hello\n";c}
```

Then, from the command line you execute:

```
>p func1();p "goodbye\n"
```

This will print *hello*, but not *goodbye*.

Types of Breakpoints

Breakpoints can be separated into two general classes:

- Individual (single) breakpoints

  These are explicitly set by the user at a given location.

- All-Procedure or All-Paragraph breakpoints

  These are breakpoints attached to all debuggable procedures or paragraphs by a single command. They do not have a count or lifespan.

There are six basic types of single breakpoints. There can only be one type of single breakpoint at a given location in the code.

Generic	Set with the *b* (*breakpoint*) command at a given source-line.
Address	Set with the *ba* (*breakpoint address*) command at a given address (which might not correspond directly to a source line).
Procedure or paragraph beginning (entry)	Set with the *bb* (*breakpoint beginning*) command at the first executable statement of a procedure.
Procedure or paragraph exit	Set with the *bx* (*breakpoint exit*) command at the common exit point of a procedure, for example, the procedure epilogue where all returns go through (usually does not correspond to a source line).
Procedure trace (entry/exit)	Set with the *bt* (*breakpoint trace*) command at the procedure entry and exit.
Uplevel	Set with the *bu* (*breakpoint uplevel*) command at the return address of a given procedure call, for example, the first instruction executed after a return (which might not correspond directly to a source line).

There are three basic types of all-procedure breakpoints. These may co-exist with other all-procedure breakpoints and/or a single breakpoint at a given location.

Procedure (beginning)	Set with the *bp* (*breakpoint procedure*) command at the first executable statement of all procedures.
Procedure exit	Set with the *bpx* command at the common exit point of all procedures.
Procedure trace	Set with the *bpt* command at the entry and exit of all procedures.

At a given procedure entry, up to four command lists can be associated with the location:

- Global breakpoint command list

  Set with the *abc* command.
- Individual procedure beginning breakpoint command list

  Set with the *bb* (*breakpoint beginning*) command.
- All-procedure beginning breakpoint command list

  Set with the *bp* (*breakpoint procedure*) command.
- All-procedure trace breakpoint command list

  Set with the *bpt* command.

Also, at a given procedure exit, up to four command lists can be associated with the location:

- Global breakpoint command list

  Set with the *abc* command.
- Individual procedure exit breakpoint command list

  Set with the *bx* (*breakpoint exit*) command.
- All-procedure exit breakpoint command list

  Set with the *bpx* command.
- All-procedure trace breakpoint command list

  Set with the *bpt* command.

There are two basic types of all-paragraph breakpoints. These may co-exist with other all-paragraph breakpoints and/or a single breakpoint at a given location.

Paragraph (beginning)	Set with the *bpg* (*breakpoint paragraph*) command at the first executable statement of all paragraphs.
Paragraph trace	Set with the *tpg* (*trace paragraph*) command at the entry of all paragraphs.

At a given paragraph entry, up to four command lists can be associated with the location:

- Global breakpoint command list

  Set with the *abc* command.
- Individual paragraph beginning breakpoint command list

  Set with the *bb* (*breakpoint beginning*) command.
- All-paragraph beginning breakpoint command list

  Set with the *bpg* (*breakpoint paragraph*) command.
- All-paragraph trace breakpoint command list

  Set with the *tpg* command.

At a given paragraph exit, up to two command lists can be associated with the location:

- Global breakpoint command list

  Set with the *abc* command.
- Individual paragraph exit breakpoint command list

  Set with the *bx* (*breakpoint exit*) command.

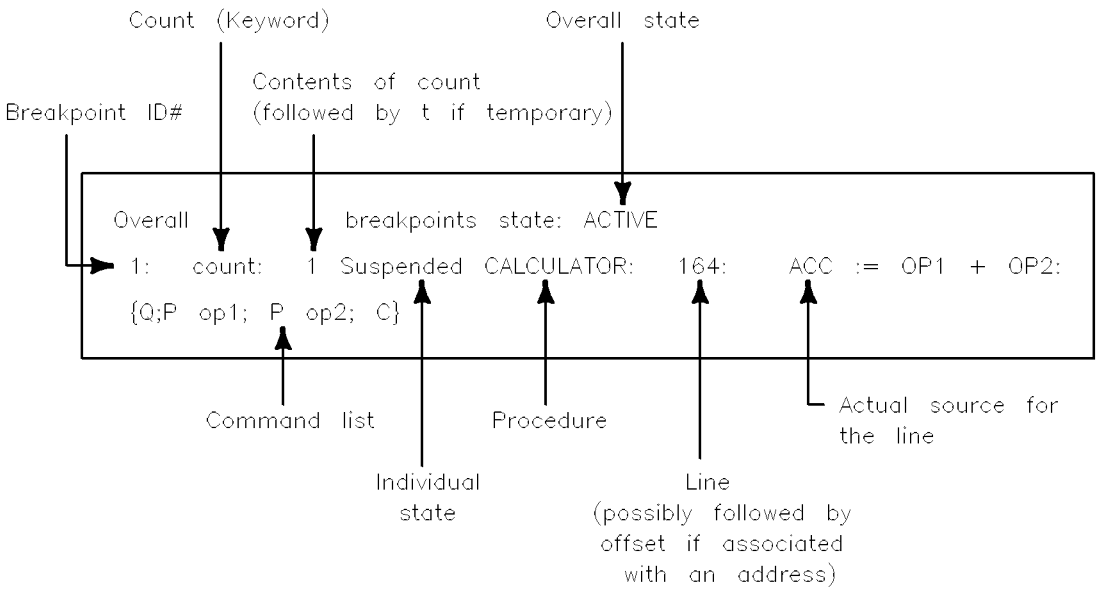
Overall Breakpoint Commands

lb (list breakpoints)

$\left\{\begin{array}{l}\texttt{lb}\\\texttt{list breakpoints}\end{array}\right\}$

Displays all breakpoints in the program, both active and suspended, and the overall breakpoint state.

The display shows the number, count, status and commands for each breakpoint. Figure 4-2 gives an example of the information that is displayed for a typical breakpoint. This information is also displayed whenever a breakpoint is added or deleted.


Count (Keyword)
Overall state
Contents of count (followed by t if temporary)
Breakpoint ID#
Command list
Procedure
Actual source for the line
Individual state
Line (possibly followed by offset if associated with an address)

```
Overall          breakpoints state: ACTIVE
1:  count:   1 Suspended CALCULATOR:  164:   ACC := OP1 + OP2:
{Q;P op1; P op2; C}
```

Figure 4-2. Listing a Breakpoint

tb (toggle breakpoints)

$$\left\{\begin{array}{l}\texttt{tb}\\ \texttt{toggle breakpoints}\end{array}\right\}$$

Toggles the overall breakpoint state from active to suspended or vice versa. The state of the individual breakpoints remains unchanged.

Breakpoint Creation Commands

b (breakpoint)

$\left\{\begin{array}{l}\texttt{b}\\\texttt{breakpoint}\end{array}\right\}$ [*location*] [\count] [*command-list*]

Sets a breakpoint at the location that you specify. If you do not enter a location, the current line in the source window is used. The breakpoint is executed on each occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list. The command list will be executed when the breakpoint is reached and its count is zero. See the definition for *location*, *count*, and *command list* at the beginning of this section, "Breakpoint Commands".

In the following example, a breakpoint is set at the current location in the source window and is executed every fourth execution of the source statement. Since there is no command list, no commands are executed when the breakpoint is reached. Instead, the debugger will just enter command mode at that point.

```
> b \4
```

To set a breakpoint in a different file, procedure, or HP COBOL II subprogram, use the *v* (*view*) command to display the file, procedure, or subprogram in the current viewing location window and search for the line on which to set the breakpoint. If you know where to set the breakpoint in another file, procedure, or subprogram enter this command with the procedure and line. For example, the following command sets a breakpoint at line 355 in procedure cmp80.

```
>b cmp80:355
```

To set a breakpoint using a label instead of a line number, enter the label name instead of the line number. For example,

```
>b cmp80#totsls
```

ba (breakpoint address)

$\left\{\begin{array}{l}\texttt{ba}\\\texttt{breakpoint address}\end{array}\right\}$ *address* [\count] [*command-list*]

Sets a breakpoint at the specified address. Note that the address can be specified by giving the name of a procedure, subprogram, or an expression containing such a name. The breakpoint is executed on each occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list. See the definition for *address* (*location*), *count*, and *command list* at the beginning of this section, "Breakpoint Commands".

The following is an example:

```
>ba printf+0x0018
Overall breakpoints state:  ACTIVE
Added:
 2: count:   1  Active    printf +0x00000018: (line unknown)
```

Caution

Be sure the address given in the *ba* (*breakpoint address*) command is a code address in the child process or errors might ensue.

bb (breakpoint beginning)

$\left\{\begin{array}{l}\texttt{bb}\\ \texttt{breakpoint beginning}\end{array}\right\}$ [*depth*] [\count] [*command-list*]

Sets a breakpoint at the first executable statement of the procedure or subprogram at the specified depth on the program stack. If you do not enter a depth, the procedure or subprogram shown in the source window is used (this might not be the same as the procedure or subprogram at depth zero in the stack).

The breakpoint is executed on the occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list. See the definitions for *count* and *command list* at the beginning of this section, "Breakpoint Commands".

bt (breakpoint trace)

$\left\{\begin{array}{l}\texttt{bt}\\ \texttt{breakpoint trace}\end{array}\right\} \left[\begin{array}{l}\textit{proc}\\ \textit{depth}\end{array}\right]$ [\count] [*command-list*]

Sets a *trace* breakpoint at the current or named procedure or subprogram or at the procedure or subprogram that is at the specified depth on the program stack. A breakpoint is set at the entry and exit point of the procedure or subprogram. The breakpoint is executed on the occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list. See the definitions for *count* and *command list* at the beginning of this section, "Breakpoint Commands".

If you include a command list, it is executed at the beginning of the procedure or subprogram. The following command list will be executed at the end of the procedure or subprogram.

```
{Q;p $ret0\d;c}
```

If you omit a command list, the following two command lists are executed at the beginning and end of the procedure or subprogram, respectively.

```
{Q; t 2; c} {Q;p $ret0\d;c}
```

The first (entry) command list above displays the two procedures at the top of the stack (the current procedure and the procedure which called it) and their parameters, then continues. The exit command list prints the return value of the procedure, then continues.

To enter a different command list for the exit point of the procedure or subprogram, use the *bx* (*breakpoint exit*) command.

bu (breakpoint uplevel)

$\left\{\begin{array}{l}\texttt{bu}\\\texttt{breakpoint uplevel}\end{array}\right\}$ [*depth*] [\count] [*command-list*]

Sets an *uplevel* breakpoint to occur immediately on return from the procedure or subprogram at the specified depth on the program stack. This command is useful for examining values returned from procedures or subprograms. For example, when execution pauses in procedure B (called from procedure A), you can set an uplevel breakpoint so that a breakpoint occurs when execution returns to procedure A.

If you omit depth, one is used (zero is the current location). The following example sets a permanent breakpoint at the current level in the stack (the current level is the value of the program counter $pc):

```
>bu 0
```

If $*pc* corresponds to the beginning of a source line, this is equivalent to:

```
>b
```

The breakpoint is executed on the occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list. See the definitions for *count* and *command list* at the beginning of this section, "Breakpoint Commands".

bx (breakpoint exit)

$\left\{\begin{array}{l}\texttt{bx}\\\texttt{breakpoint exit}\end{array}\right\}$ [*depth*] [\count] [*command-list*]

Sets an *exit* breakpoint at the epilogue code of the procedure or subprogram at the specified depth on the program stack. The breakpoint is set at a point such that all returns go through it. If you do not enter a depth, the procedure shown in the source window is used (this might not be the same as the procedure at depth zero in the stack).

The breakpoint is executed on the occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list. See the definitions for *count* and *command list* at the beginning of this section, "Breakpoint Commands".

Breakpoint Status Commands

ab (activate breakpoint)

$$\left\{\begin{array}{l}\texttt{ab}\\ \texttt{activate breakpoint}\end{array}\right\}\left[\begin{array}{l}\textit{number}\\ *\end{array}\right]$$

Activates the breakpoint having the number (ID) that you specify. If you do not enter a number, the breakpoint at the current line is activated. If there is no breakpoint at the current line, the debugger displays all the breakpoints so that you can select one to activate.

Use the asterisk (*) to activate all breakpoints including, all-procedure and all-paragraph breakpoints.

bc (breakpoint count)

$$\left\{\begin{array}{l}\texttt{bc}\\ \texttt{breakpoint count}\end{array}\right\}\textit{number expr}$$

Sets the count of the specified breakpoint number to the integer value of the evaluated expression that you enter. A negative value indicates a temporary breakpoint. A count cannot be assigned to an all-procedures or all-paragraphs breakpoint. Use the *lb* (*list breakpoints*) command to determine the number to enter.

db (delete breakpoint)

$$\left\{\begin{array}{l}\texttt{db}\\ \texttt{delete breakpoint}\end{array}\right\}\left[\begin{array}{l}\textit{number}\\ *\end{array}\right]$$

Deletes the breakpoint having the number (ID) that you specify. If you do not enter a number, the breakpoint at the current line is deleted. If the breakpoint that you specify does not exist, the debugger displays all the breakpoints so that you can select one to delete.

Use the asterisk (*) to delete all breakpoints, including all-procedure and all-paragraph breakpoints.

sb (suspend breakpoint)

$$\left\{\begin{array}{l}\texttt{sb}\\ \texttt{suspend breakpoint}\end{array}\right\}\left[\begin{array}{l}\textit{number}\\ *\end{array}\right]$$

Suspends (deactivates) the breakpoint having the number (ID) that you specify. If you do not enter a number, the breakpoint at the current line is suspended (use the *lb* (*list breakpoints*) to determine the numbers to enter). To reactivate the breakpoint use the *ab* (*activate breakpoint*) command.

Use the asterisk (*) to suspend all breakpoints, including all-procedure and all-paragraph breakpoints. This also causes the overall breakpoint state to become suspended.

All-Procedures Breakpoint Commands

bp (breakpoint procedure)

$\left\{\begin{array}{l}\texttt{bp}\\ \texttt{breakpoint procedure}\end{array}\right\}[\,\textit{command-list}\,]$

Sets permanent *procedure* breakpoints at the first executable statement of every procedure for which debugger information is available (this is equivalent to executing a *bb* (*breakpoint beginning*) for every procedure. The breakpoint is encountered each time the procedure is entered. When any entry procedure breakpoint is encountered, the command list is executed. See the definition for *command list* at the beginning of this section, "Breakpoint Commands".

This command is useful for stepping through and tracing an HP FORTRAN 77, HP Pascal, or HP C program. Refer to the *bpg* (*breakpoint paragraph*) command in this chapter for HP COBOL II programs.

The following example sets breakpoints at the beginning of each procedure. The command list causes the name of the procedure and the values of its arguments to be displayed before continuing.

```
bp {Q; t 1; c}
```

You can set other breakpoints, either permanent or temporary, at the same locations as the procedure breakpoints without replacing them. However, if an all-procedure and nonprocedure breakpoint are set at the same location, the nonprocedure breakpoint is executed first.

You cannot alter the count of a procedure breakpoint. You also cannot set or delete procedure breakpoints individually. To delete procedure breakpoints, use the *dp* command.

bpt `bpt` [*command-list*]

Sets permanent *procedure trace* breakpoints at the first and last executable statement of every procedure for which debugger information is available. The breakpoints are encountered each time the procedure is entered and exited. The command list, if any, is associated with the entry breakpoint. See the definition for *command list* at the beginning of this section, "Breakpoint Commands".

If no command list is specified, the entry command list defaults to:

```
{Q;t 2;c}
```

where:

`Q`	Is the `Quiet` command that tells the debugger not to display a breakpoint.
`t 2`	Pops the top two entries off the stack.
`c`	Is the command used to continue debugging.

The exit command list is:

```
{Q;p $ret0\d;c}
```

where:

`Q`	Is the `Quiet` command that tells the debugger not to display a breakpoint.
`p $ret0\d`	Displays the value of the special variable `ret0` as an integer value.
`c`	Is the command used to continue debugging.

You can set other breakpoints, either permanent or temporary, at the same locations as the procedure breakpoints without superceding them. However, if an all-procedure and nonprocedure breakpoint are set at the same location, the nonprocedure breakpoint is executed first.

You cannot alter the count of a procedure trace breakpoint. You also cannot set or delete procedure breakpoints individually. To delete procedure trace breakpoints, use the *dpg* command.

bpx

`bpx` [*command-list*]

Sets permanent *procedure exit* breakpoints after the last executable statement of every procedure for which debugger information is available. The breakpoint is encountered each time the procedure is exited. When any procedure exit breakpoint is encountered, the command list is executed. See the definition for *command list* at the beginning of this section, "Breakpoint Commands".

You can set other breakpoints, either permanent or temporary, at the same locations as the procedure breakpoints without superceding them. However, if an all-procedure and nonprocedure breakpoint are set at the same location, the nonprocedure breakpoint is executed first.

You cannot alter the count of a procedure exit breakpoint. You also cannot set or delete procedure exit breakpoints individually. To delete procedure exit breakpoints, use the *Dpx* command.

dp (delete procedure)

$$\left\{\begin{array}{l}\texttt{dp}\\ \texttt{delete procedure}\end{array}\right\}$$

Deletes all *procedure* breakpoints set with the *bp* (*breakpoint procedure*) command. All breakpoints set by commands other than the *bp* command will remain set.

You cannot delete procedure breakpoints individually.

Dpt

`Dpt`

Deletes all *procedure trace* breakpoints at the first and last executable statement of every procedure. All breakpoints set by commands other than the *bpt* command will remain in effect.

You cannot delete procedure trace breakpoints individually.

Dpx

`Dpx`

Deletes all *procedure exit* breakpoints at the last executable statement of every procedure. All breakpoints set by commands other than the *bpx* command will remain in effect.

You cannot delete procedure exit breakpoints individually.

Global Breakpoint Commands

abc

`abc` *command-list*

Defines a global breakpoint command list which will be executed whenever any user-defined breakpoint is encountered. This includes single, procedure, procedure trace, procedure exit, paragraph, or paragraph trace breakpoints. These commands will be executed before any commands associated with the breakpoint. See the definition for *command list* at the beginning of this section, "Breakpoint Commands".

This example suppresses the `breakpoint at` *address* message normally printed for all breakpoints.

```
>abc Q
```

dbc

`dbc`

Deletes the global breakpoint command list.

All-Paragraph Breakpoint Commands

bpg (breakpoint paragraph)

$$\left\{\begin{array}{l}\texttt{bpg}\\\texttt{breakpoint paragraph}\end{array}\right\}[\textit{command-list}]$$

Sets permanent *paragraph* breakpoints at the first executable statement of every HP COBOL II paragraph and section for which debugger information is available. The breakpoint is encountered each time the paragraph or section is entered. When any entry paragraph breakpoint is encountered, the *command list* is executed. See the definition for *command list* at the beginning of this section, "Breakpoint Commands".

This command is useful for stepping through and tracing an HP COBOL II program. Refer to the *bp* (*breakpoint procedure*) command for HP FORTRAN 77, HP Pascal, and HP C programs.

The following example sets breakpoints at the beginning of each paragraph and section. The breakpoints are traced quietly without suspending the program.

```
>bpg {Q; t 1; c}
```

You can set other breakpoints, either permanent or temporary, at the same locations as the paragraph breakpoints without superceding them. However, if a paragraph and nonparagraph breakpoint are set at the same location, the nonparagraph breakpoint is executed first.

You cannot alter the count of a paragraph breakpoint. You also cannot set or delete paragraph breakpoints individually. To delete all-paragraph breakpoints, use the *dpg* command.

dpg (delete paragraph)

$$\left\{\begin{array}{l}\texttt{dpg}\\\texttt{delete paragraph}\end{array}\right\}$$

Deletes all *paragraph* breakpoints set with the *bpg* (*breakpoint paragraph*) or *tpg* (*trace paragraph*) commands. Breakpoints set with other commands will remain in effect.

You cannot delete individual paragraph breakpoints.

tpg (trace paragraph)

$\left\{\begin{array}{l}\texttt{tpg}\\\texttt{trace paragraph}\end{array}\right\}$ [*command-list*]

Sets permanent *paragraph trace* breakpoints at the first executable statement of every HP COBOL II paragraph and section for which debugger information is available. The breakpoints are encountered each time the paragraph or section is entered. The command list, if any, is associated with the entry breakpoint. See the definition for *command list* at the beginning of this section, "Breakpoint Commands".

If no command list is specified, the entry command list defaults to:

```
{Q;t 2;c}
```

where:

`Q`	Is the `Quiet` command that tells the debugger not to display a breakpoint.
`t 2`	Pops the top two entries off the stack.
`c`	Is the command used to continue debugging.

You can set other breakpoints, either permanent or temporary, at the same locations as the trace paragraph breakpoints without superceding them. However, if a paragraph and nonparagraph breakpoint are set at the same location, the nonparagraph breakpoint is executed first.

You cannot alter the count of a trace paragraph breakpoint. You also cannot set or delete trace paragraph breakpoints individually. To delete trace procedure breakpoints, use the *Dpt* command.

This command is very similar to the *bpt* command, but differs in the following ways:

- *bpt* is targeted at HP FORTRAN 77, HP Pascal, or HP C procedures. *tpg* operates on all HP COBOL II paragraphs and sections.
- *bpt*, by default, prints results upon exiting a procedure. *tpg* does not.

Auxiliary Breakpoint Commands

Although the *any string*, *if*, and *Quiet* commands are not actually breakpoint commands, they are used almost exclusively in breakpoint and assertion command lists. Consequently, they are documented here.

"*any string*"

"*any string*"

Causes any string that is enclosed in quotation marks to be echoed to the screen. The string command is useful for labeling breakpoint output, particularly for recording a debugger session. You can include character escape sequences in the string (for example, \t). See table 4-1 "Escape Sequences" for more information.

In the following example, the "*any string*" command is used to label the display of a data-item which otherwise doesn't have a name (the debugger just prints an address in such cases). Note the use of the character escape \n (new line).

```
>"flavor_list head =>\n"; p *flavor:list->head
flavor_list head =>
0x68023004 struct {
    chocolate = 1597845365;
    tutifruti = 2.21414e-10;
}
```

i (if)

$\left\{\begin{array}{l}\texttt{i}\\\texttt{if}\end{array}\right\}$ *expr command-list* [*command-list*]

Lets you conditionally execute commands in a command list. If the expression evaluates to a non-zero value, the first group of commands is executed. If the expression evaluates to zero, the second command list, if it exists, is executed. The *i* (*if*) command can be nested in other command lists.

The following *b* (*breakpoint*) command (set at entry to procedure *proc*) uses the *i* (*if*) command to conditionally print a value only if a certain condition is true.

```
>b proc {Q; if (list->head.fld > 0) {p list->head.name}; c }
```

Q (Quiet)

$\left\{\begin{array}{l}\texttt{Q}\\\texttt{Quiet}\end{array}\right\}$

Suppresses the `breakpoint at` *address* debugger messages that are normally displayed when a breakpoint is encountered. This enables you to display variable values without cluttering the command window. The *Q* (*Quiet*) command must be the first command in a command list; otherwise, it is ignored.

Assertion Control Commands

An **assertion** is a list of one or more debugger commands that are executed before each source statement. Assertions are useful for tracing serious software defects, such as corrupt global variables, or mysterious side effects. The assertion control commands are:

- a (assert)
- aa (activate assertion)
- da (delete assertion)
- la (list assertions)
- sa (suspend assertion)
- ta (toggle assertions)
- x (exit)

Assertions can be activated or inactivated (suspended) individually. When an assertion is suspended, information for that assertion is retained, but it will not be evaluated during program execution. There is also an overall assertion mode for assertion activation and suspension which is independent of the state of any individual assertion. Any given assertion will be evaluated during program execution only if it is individually activated and the overall mode is activated.

The *if*, *Quiet* and "*any string*" commands are useful in assertion command lists. For more information about these commands, see the subsection called "Auxiliary Breakpoint Commands" in the "Breakpoint Commands" section.

Note

Assertions slow program execution because the commands for all active assertions are executed before each source statement. If you use the assertion commands in a breakpoint command list, you will be able to limit the regions of slowed execution to your actual areas of interest in the program.

a (assert)

$\left\{\begin{array}{l}\texttt{a}\\ \texttt{assert}\end{array}\right\}$ *command-list*

Creates an assertion consisting of the command list that you enter. You can enclose an assertion command list in braces to separate it from other commands on the same line. Errors in assertion command lists are not identified until the assertion is executed. If there is an error, an error message is displayed, but execution continues. Assertions, like breakpoints, are identified by a unique number assigned by the debugger. They also have an overall state, whereby all assertions can be activated or suspended as a group. Use the *la* (*list assertions*) command to see a list of assertions, their identifying numbers (ID), and the overall state.

Caution

In an assertion command list, you can use the following job control commands only after an *x* (*exit*) command, which suspends execution of the program.

- r (run)
- R (Run)
- c (continue)
- C (Continue)
- s (step)
- S (Step)
- k (kill)

Also, job control commands cannot be used in an assertion command list unless all assertions are suspended first. The following is an example of a typical command list command sequence.

```
{l; x 1; c}
```

The following examples show how to use this command.

```
a {L}
```

This "assert list" command traces program execution one line at a time until the program stops. (The program stops on normal termination, when a breakpoint is encountered or when CNTRL Y is pressed).

```
a {L; if (xyz> (def-9) *10) {ta;x 1; c} {p abc -= 10}}
```

This assertion displays the line that will be executed next, then checks the *if* statement condition. If it is true, assertion mode and all assertions are suspended, and the program continues executing. If the condition is false, the value of *abc* is decremented by 10, the next source line is executed, and the command list is executed again. The number after the *exit* command (*x 1*) enables the debugger to recognize the continue command which follows it. If just *x* or (*x 0*) was used, the remainder of the command would not be executed, and the debugger would again prompt for commands as if a breakpoint was reached. Note that the *ta* (*toggle assertions*) command is used to toggle assertions to suspend them because the *c* (*continue*) command cannot be used while assertions are active.

```
a {if (abc .NE. $abc) {p $abc = abc; if (abc .GT. 9) {x} } p abc}
```

This command list displays the value of the global variable, *abc*, and suspends program execution if the variable exceeds a certain value. $*abc* is a special variable that keeps track of when the value of *abc* changes.

Caution

If you single step or run with assertions through a call to *longjmp* (on *setjmp*(LIBC)), the child process will probably take off free-running as the debugger sets but never hits an uplevel breakpoint.

aa (activate assertion)

$$\left\{\begin{array}{l}\texttt{aa}\\ \texttt{activate assertion}\end{array}\right\}\left[\begin{array}{l}\textit{number}\\ *\end{array}\right]$$

Activates the assertion having the number (ID) that you enter. Use the *la* (*list assertions*) command to determine the number associated with an assertion. Using the * option causes all assertions to be activated.

Overall assertion mode is activated if the last suspended assertion is activated.

da (delete assertion)

$$\left\{\begin{array}{l}\texttt{da}\\ \texttt{delete assertion}\end{array}\right\}\left[\begin{array}{l}\textit{number}\\ *\end{array}\right]$$

Deletes the assertion having the number (ID) that you enter. Use the *la* (*list assertions*) command to determine the number associated with an assertion. Using the * option causes all assertions to be deleted.

la (list assertions)

$$\left\{\begin{array}{l}\texttt{la}\\ \texttt{list assertions}\end{array}\right\}$$

Lists the number, the state (active or suspended) and the command list for each assertion, as well as the overall assertion state (active or suspended).

Use this command to find the number of a particular assertion before using the *aa* (*activate assertion*), *da* (*delete assertion*) and *sa* (*suspend assertion*) commands.

The following example lists the status of two assertions:

```
Overall assertion state: ACTIVE

1:  Active     if(abc.NE.$abc){$abc = abc;p abc/d; if(abc.GT.9){x}}

2:  Suspended  L;if(xyz.GT.(def-9)*10) {ta;x 1;c} {p abc-=10}}
```

sa (suspend assertion)

$\left\{\begin{array}{l}\texttt{sa}\\\texttt{suspend assertion}\end{array}\right\}\left[\begin{array}{l}\textit{number}\\*\end{array}\right]$

Suspends the assertion having the number (ID) that you enter. Use the *la* (*list assertions*) command to determine the number associated with an assertion. Using the * option causes all assertions to be suspended.

Suspended assertions continue to exist but are not evaluated until activated again. Overall assertion mode is suspended if the last active assertion is suspended.

ta (toggle assertions)

$\left\{\begin{array}{l}\texttt{ta}\\\texttt{toggle assertions}\end{array}\right\}$

Toggles the overall assertion state between active and suspended. The overall assertion state does not affect the state of individual assertions.

x (exit)

$\left\{\begin{array}{l}\texttt{x}\\\texttt{exit}\end{array}\right\}[\textit{expr}]$

Causes program execution to stop as if a breakpoint has been reached. A message like the following will be printed:

```
Hit on assertion 1: command-list
Last line executed was:
    file: source text
Next line to execute is:
    file: source text
```

If the expression (*expr*) is not given or it evaluates to zero, the debugger returns to command mode, ignoring any remaining commands in the assertion command list. If *expr* evaluates to non-zero, any remaining commands in the command list are executed.

Note

This command can only be used in an assertion command list.

Datatrace Control Commands

A **datatrace** is used to monitor the value of one or more variables. When the value changes, commands specified in a command list are executed. Datatraces are useful for tracing serious software defects, such as corrupt global variables, or mysterious side effects. The datatrace control commands are:

- ndt
- adt (activate datatrace)
- ddt (delete datatrace)
- ldt (list datatraces)
- sdt (suspend datatrace)
- tdt (toggle datatraces)
- x (exit)

Datatraces can be activated or inactivated (suspended) individually. When a datatrace is suspended, information for that datatrace is retained, but it will not be evaluated during program execution. There is also an overall datatrace mode for datatrace activation and suspension which is independent of the state of any individual datatrace. Any given datatrace will be evaluated during program execution only if it is individually activated and the overall mode is activated.

The *if*, *Quiet* and "*any string*" commands are useful in datatrace command lists. For more information about these commands, see the subsection called "Auxiliary Breakpoint Commands" in the "Breakpoint Commands" section.

Note

Datatraces slow program execution because the value of the monitor variables are checked before each source statement. If you use the datatrace commands in a breakpoint command list, you will be able to limit the regions of slowed execution to your actual areas of interest in the program.

ndt (datatrace)

$\left\{ \begin{array}{l} \texttt{ndt} \\ \texttt{datatrace} \end{array} \right\} \textit{var} \ [\, \{ \textit{command-list} \} \, [\textit{silent}] \,]$

Creates a datatrace for the specified variable *var* consisting of the command list that you enter. You can enclose a datatrace command list in braces to separate it from other commands on the same line. Errors in datatrace command lists are not identified until the datatrace is executed. If there is an error, an error message is displayed, but execution continues. Datatraces, like breakpoints, are identified by a unique number assigned by the debugger. They also have an overall state, whereby all datatraces can be activated or suspended as a group. Use the *ldt* (*list datatraces*) command to see a list of datatraces, their identifying numbers (ID), and the overall state.

Caution

In a datatrace command list, you can use the following job control commands only after an *x* (*exit*) command, which suspends execution of the program.

- r (run)
- R (Run)
- c (continue)
- C (Continue)
- s (step)
- S (Step)
- k (kill)

Also, job control commands cannot be used in a datatrace command list unless all datatraces are suspended first. The following is an example of a typical command list command sequence.

```
{l; x 1; c}
```

The following examples show how to use this command.

```
ndt i
```

This sets a datatrace on variable `i`. The symbolic debugger will display the current value of `i` and stop when that value changes. The default command list `{L}` will be used.

```
ndt i { if(i < 10) {c} }
```

This sets a datatrace on the variable `i`, but stops execution only when the new value reaches or exceeds 10.

```
ndt i { if(i <= 10) {c} {"i is greater than 10"} } silent
```

This sets a datatrace on the variable `i`. When the value of `i` exceeds 10, the message:

```
i is greater than 10
```

is printed. The `silent` option prevents the symbolic debugger from printing the standard messages.

Caution

If you single step or run with datatraces through a call to *longjmp* (on *setjmp*(LIBC)), the child process will probably take off free-running as the debugger sets but never hits an uplevel breakpoint.

adt (activate datatrace)

$$\left\{\begin{array}{l}\texttt{adt}\\ \texttt{activate datatrace}\end{array}\right\}\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$$

Activates the datatrace having the number (ID) that you enter. Use the `ldt` (list datatraces) command to determine the number associated with a datatrace. Using the `*` option causes all datatraces to be activated.

Overall datatrace mode is activated if the last suspended data trace is activated.

ddt (delete datatrace)

$$\left\{\begin{array}{l}\texttt{ddt}\\ \texttt{delete datatrace}\end{array}\right\}\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$$

Deletes the datatrace having the number (ID) that you enter. Use the `ldt` (list datatraces) command to determine the number associated with a datatrace. Using the `*` option causes all datatraces to be deleted.

ldt (list datatraces)

$$\left\{\begin{array}{l}\texttt{ldt}\\ \texttt{list datatraces}\end{array}\right\}$$

Lists the number, the state (active or suspended) and the command list for each datatrace, as well as the overall datatrace state (active or suspended).

Use this command to find the number of a particular datatrace before using the `adt` (activate datatrace), `ddt` (delete datatrace) and `sdt` (suspend datatrace) commands.

sdt (suspend datatrace)

$\left\{\begin{array}{l}\texttt{sa}\\\texttt{suspend datatrace}\end{array}\right\}\left[\begin{array}{l}\textit{number}\\ *\end{array}\right]$

Suspends the datatrace having the number (ID) that you enter. Use the `ldt` (list datatraces) command to determine the number associated with a datatrace. Using the `*` option causes all datatraces to be suspended.

Suspended datatraces continue to exist but are not evaluated until activated again. Overall datatrace mode is suspended if the last active datatrace is suspended.

ta (toggle datatraces)

$\left\{\begin{array}{l}\texttt{tdt}\\\texttt{toggle datatraces}\end{array}\right\}$

Toggles the overall datatrace state between active and suspended. The overall datatrace state does not affect the state of individual datatraces.

x (exit)

$\left\{\begin{array}{l}\texttt{x}\\\texttt{exit}\end{array}\right\}[\,\textit{expr}\,]$

Causes program execution to stop as if a breakpoint has been reached. A message like the following will be printed:

```
Hit on datatrace 1: command-list
Last line executed was:
    file: source text
Next line to execute is:
    file: source text
```

If the expression (*expr*) is not given or it evaluates to zero, the debugger returns to command mode, ignoring any remaining commands in the datatrace command list. If *expr* evaluates to non-zero, any remaining commands in the command list are executed.

Note

This command can only be used in a datatrace command list.

Record and Playback Commands

The record and playback commands allow reproduction of an HP Symbolic Debugger session by saving debugger commands in a file, which can later be used to execute the commands. The record and playback commands are useful for finding bugs that require many debugger actions to isolate or reproduce. The *record-all* command is useful for saving a log of the entire session.

The record and playback commands do not:

- Save debugger responses to commands in the record file. An exception to this is the *record-all* command that logs all debugger output as well as user input to the debugger. Note that a *record-all* file cannot be used as a playback file.
- Record commands in command lists for breakpoints and assertions as they are executed.
- Copy command lines that begin with > , < , :, or ! to the current record file. However, this limitation can be overridden by beginning those lines with blanks.
- Record output from the user program (child process). This may be done using output redirection (>) in the *r* (*run*) command line.

The only commands recorded are those read from the keyboard or a playback file. Commands in a breakpoint or assertion command list are *not* recorded as they are evaluated.

Table 4-5 lists the record and playback commands and table 4-6 lists the record-all commands. These are used to log all of the output generated in the command window by the debugger. Output generated by the child process is *not* recorded.

Caution

Be careful not to try to play back from a file currently opened for recording or record from a file currently opened for playback. This could cause problems with your debugger session.

Record and Playback Commands

Table 4-5. Record and Playback Commands

Command	Description
>*file*	Sets or changes the record file to *file*, turns recording on, rewrites the file from the beginning, and only records commands. If *file* exists, you are asked if you want to overwrite.
>>*file*	Sets or changes the record file to *file*, turns recording on, and only records commands. All recording is appended to the existing *file*; otherwise, a new file is created.
>	Displays the recording state and the current recording file. Can also use ">>".
<*file*	Starts playback from the file.

<<*file*	Starts playback from the file using the "line-at-a-time" feature. Each command line from the playback file is shown before it is executed, and the debugger provides a list of the following commands for you to take some action:
	 command (`<cr>`,`S`, `<num>`, `C`, `Q`, or `?`):
	 You can use any of the above options as described:
	 `<cr>       execute one command line`
	 `S            skip one command line`
	 `<num>      execute number of command lines`
	 `C            continue through all playback`
	 `Q            quit playback mode`
	 `?            gives this explanation of the`
	 `above commands`

`tr`	Toggles recording; toggles the state of the record mechanism between active and suspended.
>`t`	Turns recording on. (active)
>`f`	Turns recording off. (suspended)
>`c`	Closes the record file.

Table 4-6. Commands Used to Record Debugger Output

Command	Description
>@*file*	Sets or changes the *record-all* file to *file*, rewrites from the beginning, and turns recording on. If *file* exists, you are asked if you want to overwrite. Captures all input to and output from the debugger command window, except user program output.
>>@*file*	Sets or changes the *record-all* file to *file*, and turns recording on. Appends *record-all* output to the existing *file*. Captures all input to and output from the debugger command window.
>`@`	Displays the current *record-all* state and file. Can also use ">>@".
`tr @`	Toggles the state of the *record-all* mechanism between active and suspended.
>`@t`	Turns *record-all* on.
>`@f`	Turns *record-all* off.
>`@c`	Closes the *record-all* file.

Macro Facility Commands

The macro facility allows you to substitute your own names for debugger commands or sequences of debugger commands. To do so, you simply define the text to be used as a straight replacement for the macro name. Thereafter, you can use your newly defined macro name to represent the debugger commands while inside a debugger session.

Note

Macros do not allow argument substitution and are only recognized when used where a command is valid. They cannot be used to modify debugger command syntax.

When defining a macro, replacement text is not immediately scanned for additional macro invocations. Rather, macro substitutions are performed as late as possible by HP Symbolic Debugger. This means that when a macro is referenced and has been evaluated, its replacement text is rescanned to determine if the replacement text contains any additional macros. Macros are not recognized inside character constants, strings, or comment (#) commands during command line processing.

Debugger commands can be redefined by a macro. However, redefining a debugger command does not redefine its abbreviation. Each must be redefined separately to change the meaning of both. For example, redefining the *list breakpoints* command as *bplist* has no effect on the *lb* abbreviation.

The invocation of recursive macros is trapped and terminates with an error message. Recursive macros are macros whose replacement text contains another reference to the same macro, or to a macro whose expansion eventually references the same macro. For example,

```
define a a
```

is flagged as an error.

Macros are not recognized unless the state of the macro mechanism is activated with the *tm* (*toggle macros*) command. If you want to see a list of your macros and their current state (active or suspended), use the *lm* (*list macros*) command.

def

`def` *name replacement-text*

Defines a macro substitution (user-defined command) for HP Symbolic Debugger commands. *Name* can be any string of letters or digits, beginning with a letter. *Replacement-text* can be any string of letters, blanks, tabs or other printing characters that represent one or more debugger commands. The string begins with the first non-white-space character following *name* and ends with the first RETURN. For example, executing this command:

```
>def myprint p flavor:list->head.tuttifrutti
myprint ==> p flavor:list->head.tuttifrutti
```

creates a macro called `myprint` which can be entered at the debugger prompt in the place of typing:

```
p flavor:list->head.tuttifrutti
```

Note

If a macro can be defined with the same name as a previous macro, The new definition will replace the old one, until it is undefined with the *undef* command, at which point the old definition is active.

tm (toggle macros)

$$\left\{\begin{array}{l}\texttt{tm}\\ \texttt{toggle macros}\end{array}\right\}$$

Toggles the state of the macro mechanism between active and suspended. When macros are suspended, the currently defined macros continue to exist, but are not replaced in the command line by their definitions. Additional macros can be defined while the macro state is suspended.

undef

$$\texttt{undef}\left\{\begin{array}{l}\mathit{name}\\ *\end{array}\right\}$$

Removes the macro defined by *name*. Using the * option causes all macros to become undefined.

Miscellaneous Commands

The miscellaneous commands perform a variety of individual tasks. The miscellaneous commands are:

- !
- :
- #
- RETURN
- ~
- am (activate more)
- debug
- f (format)
- g (goto)
- h (help)
- q (quit)
- sm (suspend more)
- tc (toggle case)

!

`!` [*MPE command*]

Escapes out of the debugger into the operating system. If a command is specified, it is automatically executed. Otherwise, a session is invoked and must be explicitly ended before the debugger can resume. When you execute the *!* command interactively, return to the debugger by hitting the RETURN key. When you use this command in an assertion or breakpoint command list, control returns to the debugger automatically.

A command can be enclosed in braces ({}) to delimit it from debugger commands on the same line. For example:

```
b 14 {!{SHOWTIME}; continue}; trace; list assertions
```

If you use the escape without giving a list of commands, you are given a colon prompt. You can now execute any MPE/iX operating system command. You can return to the debugger by typing `exit` at the colon prompt.

Note

It is recommended that you return to the debugger when finished with your session.

This command is synonymous with the *:* command.

: : [*MPE command*]

Escapes out of the debugger into the operating system. If a command is specified, it is automatically executed. Otherwise, a session is invoked and must be explicitly ended before the debugger can resume. When you execute the *:* command interactively, return to the debugger by hitting the RETURN key. When you use this command in an assertion or breakpoint command list, control returns to the debugger automatically.

A command can be enclosed in braces ({}) to delimit it from debugger commands on the same line. For example:

```
b 14 {:{SHOWTIME}; continue}; trace; list assertions
```

If you use the escape without giving a list of commands, you are given a colon prompt. You can now execute any MPE/iX operating system command. You can return to the debugger by typing `exit` at the colon prompt.

Note It is recommended that you return to the debugger when finished with your session.

This command is synonomous with the *!* command.

[*text*]

Causes the text to be interpreted as a comment. This command can be used to document the contents of record and playback files. The number symbol (#) must be the first nonblank character on the line. The rest of the line is treated as a comment and is written to the record file if the recording is on. Otherwise, it is ignored.

RETURN RETURN

Repeats the previous command. You can use this command with the following commands:

- +
- -
- p (print)
- v (view)
- s (step)
- S (Step)

This command is synonomous with the ~ command.

~ ~

Repeats the previous command. You must use the RETURN key after typing the ~. You can use this command with the following commands:

- +
- -
- p (print)
- v (view)
- s (step)
- S (Step)

This command is synonomous with the RETURN command, but is more "visible" in a record or playback file.

am (activate more)

```
{ am            }
{ activate more }
```

Activates (enables) the *more* feature. (Active is the initial state). When activated, all command window output following a debugger command is presented to you a window-full at a time, and you are prompted before displaying successive windows.

Use one of the following commands to continue.

SPACE BAR	Displays one more window-full.
RETURN	Displays one more line.
`q`	Quits scrolling and ignores the rest of the output until another debugger prompt is issued.

To view command window output in a continuous stream, use the *sm* (*suspend more*) command to suspend the `more` feature. CNTL S may be used to temporarily suspend scrolling when the `more` feature is suspended. Use CNTL Q to continue scrolling.

Note Output from the child process (program being debugged) also appears in the command window, but it is *not* controlled by the `more` feature.

debug

```
debug
```

Transfers control to the MPE NMdebugger by causing the child process to call the "DEBUG" entry point. When you exit, control returns to the HP Symbolic Debugger.

f (format)

$\left\{\begin{array}{l}\texttt{f}\\ \texttt{format}\end{array}\right\}$ ["*printf-style-format*"]

Sets the printing format used by the debugger to print an address. Only the first 19 literal and formatting characters are used (see the section on *printf* in the *HP C/XL Library Reference Manual* for a discussion of valid formats). If the format is set incorrectly, an error message appears.

Using the *f* (*format*) command without an argument will reset the format to the default format: 8 hexadecimal digits, preceded by "0x".

Note

This command is generally not needed for typical debugger use.

Caution

If you set the address printing format to something *printf* does not like, you might get an error (usually memory fault) each time you try to print an address, until you fix the format with another *f* (*format*) command.

g (goto)

$\left\{\begin{array}{l}\texttt{g}\\ \texttt{goto}\end{array}\right\}\left\{\begin{array}{l}\textit{line}\\ \#\textit{label}\end{array}\right\}$

Moves the current point of execution suspension to the specified line or label. The specified line or label must be within the same procedure (or HP COBOL II paragraph) where execution is currently suspended (at depth zero on the stack). This is not necessarily in the procedure currently being viewed. The program counter will change so that the given line number or the line that *#label* appears on becomes the next executable line. Execution does not automatically resume.

Note

For purposes of this command, the main program is treated as a procedure.

h (help)

$$\left\{\begin{array}{l}\texttt{h}\\ \texttt{help}\end{array}\right\}$$

Prints a command summary, called the *Help* file which describes the syntax and use of each command. This facility references the short form of the command only, not the long form.

The *more* facility can be used to view the file. When activated, all command window output following a debugger command is presented to you a window-full at a time, and you are prompted before displaying successive windows.

Use one of the following commands to continue.

SPACE BAR	Displays one more window-full.
RETURN	Displays one more line.
`q`	Quits scrolling and ignores the rest of the help information.

To view help information in a continuous stream, use the *sm* (*suspend more*) command to suspend the `more` feature. CNTL S may be used to temporarily suspend scrolling when the `more` feature is suspended. Use CNTL Q to continue scrolling.

q (quit)

$$\left\{\begin{array}{l}\texttt{q}\\ \texttt{quit}\end{array}\right\}$$

Quits the debugger and asks for confirmation: enter `y` (`yes`) or `n` (`no`). This command returns control to the operating system and terminates the debugging session. All files are closed and the terminal is restored to a normal mode.

sm (suspend more)

$$\left\{\begin{array}{l}\texttt{sm}\\ \texttt{suspend more}\end{array}\right\}$$

Suspends the *more* feature and lets you view the output in a continuous stream. CNTL S and CNTL Q can be used to temporarily suspend scrolling.

Use this command when you do not want the debugger to pause at the end of each window of output waiting for a continuation command. This command is particularly useful for viewing a large amount of output containing many breakpoints. To view the command window output one window-full at a time, use the *am* (*activate more*) command to activate the `more` feature.

tc (toggle case)

$\left\{\begin{array}{l}\texttt{tc}\\ \texttt{toggle case}\end{array}\right\}$

Toggles case sensitivity; determines whether or not searches or names are case sensitive (Initially, they are case insensitive.) This command affects file and procedure names, variables, and search strings used with the */* or *?* commands.

Note

Case insensitive searches equate some non-letters with other non-letters. For example, `[` and `{` are equal, as are `@` and ‘.

do

`do` [*cmdid*] [`,`*editstring*]

Re-executes the command identified by *cmdid* after applying *editstring*. The optional *cmdid* can be a positive or negative number or a string that will be searched in the history stack. If no parameters are specified, `do` executes the most recently executed command.

redo

`redo` [*cmdid*] [`,`*editstring*]

Allows you to edit and re-execute the command identified by *cmdid*. The optional *cmdid* can be a positive or negative number or a string that will be searched in the history stack. If no parameters are specified, `redo` allows you to edit the most recently executed command.

listredo

`listredo` [*start*] [`,`*end*]

Lists commands from the redo stack between *start* and *end* inclusive. The optional *start* and *end* can be positive or negative numbers.

A

Messages

This appendix lists messages that you may encounter while using HP Symbolic Debugger. Self-explanatory messages and those which relate to syntax errors, such as missing or extraneous characters in commands, are not listed in this appendix.

To assist you in finding the solution to a problem, several messages may be displayed. Look up each message in this appendix to get complete information about the action to take.

Messages are preceded by unique reference numbers that indicate the error type. Messages, with their message reference numbers, are listed in this order:

`UE300-UE785`	User Errors
`DB1-DB11`	Debugger Errors

Internal error messages, which are in the range of `IE500` to `IE825`, should not occur with normal debugger use. If they do occur, report them to your HP representative.

Child process (program) errors result in signals which are communicated to the debugger. If a program error occurs while executing a procedure call from the command line, it is handled like any other error (in other words, you can investigate the called procedure). To recover from this, or to abort a procedure call from the command line, press CTRL Y.

The following example message has a reference number of `UE313` and is listed below as it appears in this appendix:

```
UE313      MESSAGE       Invalid breakpoint type "TEXT"
```

A list of abbreviations that are used throughout this appendix and their meanings follow. Note that in all explanations, commands are given in long form, but the short form may also be used. See the chapter "HP Symbolic Debugger Commands" for further details.

ABBREVIATION	DEFINITION
CMD	A debugger command.
FILE	The name of a file.
NAME	The name of a data object.
NUM	A number.
PROC	A user program or procedure name.
TEXT	A text string; arbitrary user input.
UE*nnn*	User-created error.
DB*nnn*	A debugger error.

User Errors (UE300 - UE785)

User errors result from entering incorrect commands or from using the commands incorrectly. User errors cause the command that you entered to fail. You must correct the cause of the error and re-enter the command.

UE300	MESSAGE	`Attempt to read on non-word boundary`
	CAUSE	The debugger cannot read on a non-word aligned address.
	ACTION	Do not try to read at a non-word boundary. An incorrect reference to a data item has probably been made. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.
UE301	MESSAGE	`Attempt to write to odd address`
	CAUSE	An attempt to write a value on a non-word or half-word boundary was made.
	ACTION	Do not try to write to an odd address. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.
UE302	MESSAGE	`Address not found`
	CAUSE	The address is part of a command and is invalid. It is probably out of range.
	ACTION	Check the validity of the address and re-enter the command.
UE303	MESSAGE	`Cannot read that location`
	CAUSE	Access to the child process failed, possibly caused by an invalid address.
	ACTION	Check the validity of the address and re-enter the command.
UE304	MESSAGE	`No child process`
	CAUSE	The debugger attempted an operation that required a child process that does not exist (was not running).
	ACTION	To start a child process, use any of the `r` (`run`) or `s` (`step`) commands.

UE305	MESSAGE	`No child process AND no corefile`
	CAUSE	The debugger attempted an operation that required a child process or a core file.
	ACTION	Start a child process using any of the `r` (`run`) or `s` (`step`) commands, or restart the debugger on a valid core file.
UE306	MESSAGE	`Attempt to write to non-word boundary.`
	CAUSE	The debugger cannot write to a non-word aligned address.
	ACTION	Do not try to write to a non-word boundary. An incorrect reference to a data item has probably been made. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.
UE307	MESSAGE	`Cannot write that location`
	CAUSE	Access to a child process failed; this may have been caused by an invalid address.
	ACTION	Check the validity of the address and re-enter the command.
UE308	MESSAGE	`Bad access to child process`
	CAUSE	Failed to read data from or write data to a child process. This may have been caused by an invalid address (for example, dereferencing an invalid pointer), or by an attempt to place a breakpoint in an unwritable child process code space. Other possible causes: ■ The executable file is already being debugged in a different debugging session. ■ The process you were debugging exec'ed a different process.
	ACTION	Check the validity of the data and re-enter the command. You can also: ■ Kill the other debugging session. ■ If you need to debug the new process, adopt it with the `-P` option.

UE310	MESSAGE	`Can't set breakpoint (invalid address)`
	CAUSE	The address of the specified breakpoint command was invalid or unknown.
	ACTION	Re-enter the breakpoint command with a correct address or location.
UE311	MESSAGE	`Stack isn't that deep`
	CAUSE	The debugger tried to set a breakpoint or view a procedure at an invalid depth. The child process stack was not that deep.
	ACTION	Use the trace command to list the child process stack. This will show you how deep the stack is and what procedure is at each depth on the stack.
UE312	MESSAGE	`No symbols for that procedure`
	CAUSE	The debugger tried to set a breakpoint using a stack depth, when the procedure at that stack depth was non-debuggable.
	ACTION	Try setting a `ba` (`breakpoint address`) using the name of the procedure; for example, `ba xxx`.
UE313	MESSAGE	`Invalid breakpoint type "TEXT"`
	CAUSE	TEXT was an invalid breakpoint type.
	ACTION	Refer to the "Breakpoint Commands" section in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see valid *breakpoint* commands.
UE314	MESSAGE	`Invalid command list, must be enclosed in {}`
	CAUSE	The command list associated with a breakpoint or an assertion must be enclosed in {}.
	ACTION	Re-enter the breakpoint or assertion with the correct syntax.

UE315	MESSAGE	`Invalid line number on "breakpoint" command`
	CAUSE	The quantity given for a line number on a breakpoint command was an invalid numeric expression.
	ACTION	Re-enter the command with a valid expression.
UE319	MESSAGE	`Invalid line number on "CMD" command`
	CAUSE	The quantity given for a line number on a `b` (`breakpoint`), `v` (`view`), or `c` (`continue`) command, was an invalid numeric expression.
	ACTION	Re-enter the command with a valid expression.
UE321	MESSAGE	`Procedure "PROC" not found where specified`
	CAUSE	The nesting of procedure PROC was not properly specified.
	ACTION	Use the trace command to list the stack and find where PROC is located.
UE323	MESSAGE	`No count given for "breakpoint CMD" command`
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the "Breakpoint Commands" section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.

UE324	MESSAGE	`No count given for "breakpoint" command`
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the "Breakpoint Commands" section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.

UE325	MESSAGE	`No count given for "breakpoint address" command`
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the "Breakpoint Commands" section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.

UE326	MESSAGE	`No count given for "breakpoint beginning" command`
	CAUSE	The user failed to specify a breakpoint count (after the \) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the "Breakpoint Commands" section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.

UE327	MESSAGE	`No count given for "breakpoint count" command`
	CAUSE	The user failed to specify a breakpoint count (after the `\`) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the “Breakpoint Commands” section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.
UE328	MESSAGE	`No count given for "breakpoint trace" command`
	CAUSE	The user failed to specify a breakpoint count (after the `\`) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the “Breakpoint Commands” section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.
UE329	MESSAGE	`No count given for "breakpoint uplevel" command`
	CAUSE	The user failed to specify a breakpoint count (after the `\`) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the “Breakpoint Commands” section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.

UE330	MESSAGE	`No count given for "breakpoint exit" command`
	CAUSE	The user failed to specify a breakpoint count (after the `\`) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the "Breakpoint Commands" section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.
UE331	MESSAGE	`No count given for "CMD" command`
	CAUSE	The user failed to specify a breakpoint count (after the `\`) for a breakpoint command. Or, an attempt was made to use the `bc` (`breakpoint count`) command on an existing breakpoint.
	ACTION	Refer to the "Breakpoint Commands" section in the Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for breakpoint commands.
UE332	MESSAGE	`Count must be positive or negative`
	CAUSE	A count of zero was given for a `b` (`breakpoint`) or `bc` (`breakpoint count`) command.
	ACTION	Re-enter the command with a non-zero count.
UE333	MESSAGE	`Must specify a macro name`
	CAUSE	The `def` command was entered without arguments.
	ACTION	Refer to the "Macro Facility Commands" section in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see the correct syntax for the `def` command.

UE335	MESSAGE	`Must specify which macro to delete`
	CAUSE	The `undef` command was entered to delete or undefine a macro without giving the name of the macro to delete.
	ACTION	Use the `lm` (`list macros`) command to list all defined macros.

UE336	MESSAGE	`Unknown name or command "NAME"`
	CAUSE	An unrecognized string (NAME) was encountered as a debugger command.
	ACTION	Refer to the *HP Symbolic Debugger/iX Quick Reference* to see tables of valid debugger commands.

UE337	MESSAGE	`Unknown command "CMD"`
	CAUSE	An unrecognized string (CMD) was encountered as a debugger command.
	ACTION	Refer to the *HP Symbolic Debugger/iX Quick Reference* to see tables of valid debugger commands.

UE339	MESSAGE	`Empty assertion not added`
	CAUSE	The assertion command was given without an associated command list.
	ACTION	Re-enter the command and include a command-list within braces (`{ }`).

UE341	MESSAGE	`No breakpoint set at current location`
	CAUSE	An attempt was made to activate, delete, or suspend a breakpoint where no breakpoint was defined.
	ACTION	Use the `lb` (`list breakpoints`) command to see where breakpoints are set.

UE342	MESSAGE	`Address is required after "breakpoint address"`
	CAUSE	The `ba` (`breakpoint address`) command must be followed by a code address.
	ACTION	Use a valid code address (symbolic or numeric) with the command.

UE343	MESSAGE	`Address is required after "CMD"`
	CAUSE	The breakpoint command must be followed by a code address.
	ACTION	Use a valid code address (symbolic or numeric) with the command.
UE344	MESSAGE	`Invalid depth given for "breakpoint CMD" command`
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE345	MESSAGE	`Invalid depth given for "breakpoint beginning" command`
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE346	MESSAGE	`Invalid depth given for "breakpoint trace" command`
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE347	MESSAGE	`Invalid depth given for "breakpoint uplevel" command`
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.

UE348	MESSAGE	`Invalid depth given for "breakpoint exit" command`
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE349	MESSAGE	`Invalid depth given for "CMD" command`
	CAUSE	An attempt was made to specify a depth that is not a number greater than or equal to 0.
	ACTION	Re-enter the appropriate command with a valid depth.
UE350	MESSAGE	`Depth must be an integer`
	CAUSE	An attempt was made to specify a stack depth that is not a number.
	ACTION	Re-enter the command and specify an integer depth.
UE355	MESSAGE	`Must specify which assertion to delete`
	CAUSE	The number of the assertion to delete was not specified.
	ACTION	Use the `la` (`list assertions`) command to find the number of the assertion to delete.
UE358	MESSAGE	`Invalid expression for depth on "View" command`
	CAUSE	The `View` command was given with an expression for a depth that the debugger cannot evaluate.
	ACTION	Use the `t` (`trace`) command to view the stack for the proper procedure and depth.

UE359	MESSAGE	`Invalid expression for depth on "V" command`
	CAUSE	The `V` command was given with an expression for a depth that the debugger cannot evaluate.
	ACTION	Use the `t` (`trace`) command to view the stack for the proper procedure and depth.
UE364	MESSAGE	`Missing "{"`
	CAUSE	The `i` (`if`) command did not have a brace (`{`) following the conditional expression. Or, the expression might have been entered incorrectly.
	ACTION	Re-enter the expression, enclosing the command-lists in braces.
UE368	MESSAGE	`Map is not supported`
	CAUSE	Your version of the debugger does not support the `M` (`Map`) command, because core files are unsupported.
	ACTION	Do not enter the `M` (`Map`) command.
UE369	MESSAGE	`Unknown name "NAME"`
	CAUSE	An unrecognized string (procedure or variable name) was encountered in an expression.
	ACTION	Use the `lp` (`list procedures`), `lg` (`list globals`), `l` (`list`), `lc` (`list commons`), or `ll` (`list labels`) command to list all known procedures, globals, locals, commons, or labels.
UE372	MESSAGE	`Must specify which assertion to suspend`
	CAUSE	The number of the assertion to suspend was not specified.
	ACTION	Use the `la` (`list assertions`) command to find the number of the assertion to suspend.

UE373	MESSAGE	`Invalid expression given for "suspend assertion" command`
	CAUSE	The `sa` (`suspend assertion`) command was given with an expression that the debugger cannot evaluate.
	ACTION	Use an expression which evaluates to a number.

UE374	MESSAGE	`Invalid expression given for "sa" command`
	CAUSE	The `sa` (`suspend assertion`) command was given with an expression that the debugger cannot evaluate.
	ACTION	Use an expression which evaluates to a number.

UE375	MESSAGE	`Bad magic number NUM`
	CAUSE	The file you are trying to debug is not a valid executable file.
	ACTION	Specify a valid executable file for the program to be debugged.

UE378	MESSAGE	`Invalid expression given for "step" command`
	CAUSE	A non-numeric expression was entered as part of the `s` (`step`) command.
	ACTION	Re-enter the command with a correct numeric expression.

UE379	MESSAGE	`Invalid expression given for "Step" command`
	CAUSE	A non-numeric expression was entered as part of the `S` (`Step`) command.
	ACTION	Re-enter the command with a correct numeric expression.

UE380	MESSAGE	`Invalid expression given for "CMD" command`
	CAUSE	A non-numeric expression was entered as part of the `s` (`step`), `S` (`Step`), `t` (`trace`), `T` (`Trace`), or `sa` (`suspend assertion`) command.
	ACTION	Re-enter the command with a correct numeric expression.
UE382	MESSAGE	`Invalid expression given for "trace" command`
	CAUSE	A non-numeric expression was entered as part of the `t` (`trace`) command.
	ACTION	Re-enter the command with a correct numeric expression.
UE383	MESSAGE	`Invalid expression given for "Trace" command`
	CAUSE	A non-numeric expression was entered as part of the `T` (`Trace`) command.
	ACTION	Re-enter the command with a correct numeric expression.
UE384	MESSAGE	`Invalid window size`
	CAUSE	The numeric expression given for the new window size on the window command was not a valid numeric expression or was outside a range that is acceptable for you screen size.
	ACTION	Re-enter the command with a valid numeric expression within the range of 1 to the number of lines on your screen minus 3.
UE387	MESSAGE	`Invalid expression for mode on "exit" command`
	CAUSE	The `x` (`exit`) command was given with an expression for mode that the debugger could not evaluate.
	ACTION	Replace the mode expression with a valid numeric expression.

UE388	MESSAGE	`Invalid expression for mode on "x" command`
	CAUSE	The `x` (`exit`) command was given with an expression for mode that the debugger could not evaluate.
	ACTION	Replace the mode expression with a valid numeric expression.
UE390	MESSAGE	`Unknown name or command "CMD"`
	CAUSE	Your command is not recognized by the debugger.
	ACTION	Enter a valid debugger command.
UE391	MESSAGE	`No playback name specified`
	CAUSE	The file name is missing in a playback command.
	ACTION	Re-enter the playback command with a valid playback file name.
UE392	MESSAGE	`Can't open FILE as playback file`
	CAUSE	FILE does not exist or is unreadable.
	ACTION	Enter a valid file name, or change the file permission if it exists already.
UE393	MESSAGE	`Can't open FILE as record file`
	CAUSE	You don't have write permission in the specified directory, or a non-writable file with the same name already exists.
	ACTION	Enter a different file name, remove the old file, or change the write permission for the directory.
UE394	MESSAGE	`Operand stack overflow`
	CAUSE	An expression was too complicated for the expression handler to parse. A combination of more than 15 nested parentheses and/or pending operators may be the cause.
	ACTION	Re-enter the expression, using less than 15 nested parentheses.

UE396	MESSAGE	`Data too big to put in the child process`
	CAUSE	A string constant or other data was larger than the total size of the buffer in `xdbend.lib.sys`.
	ACTION	Re-enter a smaller string constant or data item, if applicable.
UE397	MESSAGE	`Can't store into a constant`
	CAUSE	The left side of an assignment statement was found to be a constant; it cannot be modified.
	ACTION	Use the `\t` display format for information on the assigned variable.
UE399	MESSAGE	`String too long for assignment`
	CAUSE	An attempt was made to assign a string over 1024 bytes to an HP FORTRAN 77 CHAR*, HP Pascal string, or HP Pascal packed array of char.
	ACTION	Use the `\t` display format for type information of the string assigning to, and re-enter the command with an appropriately sized string.
UE400	MESSAGE	`Incompatible operands for string assignment`
	CAUSE	An attempt was made to assign to an HP FORTRAN 77 CHAR*, HP Pascal string, or HP Pascal packed array of char, something other than an HP FORTRAN 77 CHAR*, HP Pascal string, HP Pascal packed array of char, a string constant, or a character constant.
	ACTION	Re-enter the command with a proper assignment.
UE402	MESSAGE	`Can't take the address of a constant`
	CAUSE	The operand of a `&`, `$addr`, or `addr` operator is marked as a constant type.
	ACTION	Use the `\t` display format to find the type of the operand.

UE403	MESSAGE	`Can't take the address of a register`
	CAUSE	The operand of a `&`, `$addr`, or `addr` operator is marked as a register type.
	ACTION	Use the `\t` display format to find the type of the operand.
UE404	MESSAGE	`Prefix "++" not supported`
	CAUSE	An attempt was made to use an unsupported `++` prefix operator.
	ACTION	Make sure there is a space between a `+` and a unary `+` operator (for example `2+ +5`). `+=1` can be used to increment.
UE405	MESSAGE	`Prefix "--" not supported`
	CAUSE	An attempt was made to use an unsupported `--` prefix operator.
	ACTION	Make sure there is a space between a `-` and a unary `-` operator (for example `2- -5`). `-=1` can be used to decrement.
UE406	MESSAGE	`Invalid combination of operator and operands`
	CAUSE	The debugger tried to perform a numeric operation on one or more non-numeric operands.
	ACTION	Re-enter the command with a valid expression.
UE407	MESSAGE	`Unknown operator (NUM)`
	CAUSE	An unsupported operator, with internal value NUM, was pushed on the operator stack.
	ACTION	Re-enter the command using an operator known to the current language or reset `$lang` to the language in which the operator is valid.

UE408	MESSAGE	`Misformed expression`
	CAUSE	An expression was entered incorrectly. The debugger attempts to show you where the error was detected in the command line. The error token might be one token beyond the actual error.
	ACTION	Re-enter the expression using operators and operands known to the current language or reset `$lang` to the language in which the operator or operand is valid.
UE409	MESSAGE	`Two operators in a row`
	CAUSE	The expression handler detected an improper construct in an expression.
	ACTION	Re-enter the command with a valid expression.
UE410	MESSAGE	`Postfix "++" not supported`
	CAUSE	An attempt was made to use an unsupported `++` postfix operator.
	ACTION	Make sure there is a space between a `+` and a unary `+` operator (for example `2+ +5`). `+=1` can be used to increment.
UE411	MESSAGE	`Postfix "--" not supported`
	CAUSE	An attempt was made to use an unsupported `--` postfix operator.
	ACTION	Make sure there is a space between a `-` and a unary `-` operator (for example `2- -5`). `-=1` can be used to decrement.
UE412	MESSAGE	`FORTRAN variable not pure array`
	CAUSE	An attempt was made to dereference an array that had pointer or function qualifiers, while the current language was set to FORTRAN, which does not support them.
	ACTION	Try again with `$lang` set to a different language.

UE413	MESSAGE	`Invalid real number`
	CAUSE	The specified numeric expression was not a real number.
	ACTION	See the appropriate language reference manual, or Table 4-3 in this manual, for the format of real numbers.
UE414	MESSAGE	`Misformed global name`
	CAUSE	A `:` or `::` must be followed by a variable name (string).
	ACTION	Refer to the "Entering Variable Names" section in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see how to specify global variables.
UE415	MESSAGE	`Unknown global`
	CAUSE	The variable specified with `:var` was not a recognized global variable name.
	ACTION	Use the `lg` (`list globals`) command to list all known global variables.
UE416	MESSAGE	`Need a ":" after the number`
	CAUSE	In specifying a variable, `proc:depth:var` was entered incompletely (`:var` was missing).
	ACTION	Refer to the `l` (`list`) command listing in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see a list of valid expression for variables.
UE417	MESSAGE	`Invalid local name`
	CAUSE	In specifying a variable, *proc*`[:`*depth*`]:`*var* was entered incorrectly. The variable *var* must be a valid variable name in the specified procedure, at the specified depth.

UE418	MESSAGE	`Unknown local`
	CAUSE	The variable specified with *proc*`[:`*depth*`]:`*var* was not a recognized local variable of *proc*.
	ACTION	Use the `l` (`list`) command to list all known local variables of the current *proc*, or use the `T` (`Trace`) command to list the locals, variables, and procedures on the stack.
UE419	MESSAGE	`Procedure "PROC" not found at stack depth NUM`
	CAUSE	In *proc*`:`*depth*, the procedure PROC was not on the child process stack at depth NUM. Either the stack was not that deep, or the procedure at that depth was not PROC.
	ACTION	Use the `t` (`trace`) command to list the stack.
UE420	MESSAGE	`Unknown language`
	CAUSE	An attempt was made to modify the current language by assigning an invalid language designator to the special variable `$lang`. The valid language designators are `COBOL`, `Pascal`, `FORTRAN`, `C` and `default`.
	ACTION	Re-enter the command with `COBOL`, `Pascal`, `FORTRAN`, `C` or `default` as the designator.
UE421	MESSAGE	`Local is not active`
	CAUSE	A local variable name was recognized but the procedure it belongs to was not currently active on the child process stack.
	ACTION	Re-enter the command after its procedure has been called.
UE422	MESSAGE	`Two operands in a row`
	CAUSE	The expression handler detected an improper construct in an expression.
	ACTION	Refer to the “Entering Expressions” section in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide*.

UE423	MESSAGE	`No source file for current address`
	CAUSE	The given child process address did not map to a known, debuggable source file.
	ACTION	Use the `lf` (`list files`) command to view the files the debugger recognizes, and re-enter the command with an appropriate address expression.
UE424	MESSAGE	`No search pattern`
	CAUSE	The search command (`/`, `?`, `n` (`next`), or `N` (`Next`)) was given without a search pattern (in the case of `n` (`next`) and `N` (`Next`), the previous search command `/` or `?` was provided without a pattern).
	ACTION	Refer to the individual command listings in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* for more information about search commands.
UE425	MESSAGE	`No match for "TEXT"`
	CAUSE	The search pattern (TEXT) for the `/`, `?`, `n`, (`next`) or `N` (`Next`) command was not found in the current viewing file. Note that the pattern is a literal, not a regular expression.
	ACTION	Try another pattern or view another file and search for the pattern.
UE426	MESSAGE	`Invalid display format "TEXT"`
	CAUSE	Given the data display format, or a portion of it, the TEXT contained invalid syntax.
	ACTION	Refer to Table 4-3 in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see valid data viewing formats.
UE427	MESSAGE	`Format is missing "\"`
	CAUSE	Because the command ends with a `\`, the debugger expects a format.
	ACTION	Re-enter the command with a format or without the ending `\`.

UE428	MESSAGE	`Length not allowed with "TEXT" format`
	CAUSE	Given the data display format, the TEXT did not allow the data length specification because it is irrelevant or implicit in the format.
	ACTION	Refer to Table 4-3 in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* to see valid data viewing formats.
UE429	MESSAGE	`This does not appear to be a record or union`
	CAUSE	The debugger tried and failed to dump the contents of a data object that was not a record or union.
	ACTION	Use the `\t` display format for more information.
UE430	MESSAGE	`This does not appear to be a struct or union`
	CAUSE	The debugger tried and failed to dump the contents of a data object that was not a `struct` or `union`.
	ACTION	Use the `\t` display format for more information.
UE431	MESSAGE	`No count given for b command`
	CAUSE	The debugger expected a breakpoint count after the `\`.
	ACTION	Re-enter the command with a breakpoint count, or with no `\`.
UE433	MESSAGE	`No current procedure`
	CAUSE	The debugger tried to list locals for the current viewing procedure when the procedure was undefined.
	ACTION	Use the `lp` (`list procedures`) command to list all the debuggable procedures.

UE434	MESSAGE	`No such procedure "PROC"`
	CAUSE	An attempt to list locals of a non-existent, or non-debuggable procedure PROC was made.
	ACTION	Use the `lp` (`list procedures`) command to list all known debuggable procedures.
UE435	MESSAGE	`Unrecognized "l" command`
	CAUSE	The `l` (`list`) command was given with a second part that was neither a known procedure name, nor a valid option.
	ACTION	Refer to the `l` (`list`) command listing in Chapter 4 of the *HP Symbolic Debugger/iX User's Guide* for more information.
UE438	MESSAGE	`Exiting command line procedure call`
	CAUSE	The command line procedure call environment terminated for an unusual reason, such as encountering a breakpoint during program execution, or an error was reached before the procedure was called.
	ACTION	Check the procedure call for errors and re-enter the command line procedure call.
UE439	MESSAGE	`Can't pass more than NUM arguments to called procedure`
	CAUSE	A large limit (NUM) exists on how many parameters can be passed to a procedure called from the command line.
	ACTION	Check the number of parameters for the procedure you are attempting to call. If the limit (NUM) is less than the number of parameters in the procedure, that procedure cannot be called from the command line.
UE440	MESSAGE	`Argument list too long`
	CAUSE	Arguments to the run command exceeded 1024 bytes.
	ACTION	Re-enter the run command with fewer arguments.

UE441	MESSAGE	`Can't goto a location in another procedure`
	CAUSE	The line number given to the `g` (`goto`) command was not an executable source line in the top procedure on the child process stack. This is not always the same as the current viewing procedure.
	ACTION	Re-enter the `g` (`goto`) command with a line number within the procedure on the top of the child process stack.
UE443	MESSAGE	`Signal actions are "i", "r", "s", "Q"`
	CAUSE	An invalid signal action was given.
	ACTION	Re-enter the command with a valid action: `i` (`ignore`), `r` (`report`), `s` (`stop`), or `Q` (quietly change signal action).
UE444	MESSAGE	`Unknown name`
	CAUSE	An unrecognized string (procedure or variable name) was encountered in an expression.
	ACTION	Use the `lp` (`list procedures`), `lg` (`list globals`), `l` (`list`), `lc` (`list commons`), or `ll` (`list labels`) command to list all known procedures, globals, locals, commons, or labels.
UE445	MESSAGE	`It appears that there's no debugging information in FILE`
	CAUSE	The program you are trying to debug doesn't contain debug information.
	ACTION	Recompile the program with the appropriate info-string debugging (e.g., info = `-g`), or debug the program at the assembly language level.
UE446	MESSAGE	`Misformed hex number`
	CAUSE	0x or 0X was given without digits following.
	ACTION	Re-enter the command with a valid hexadecimal number.

UE447	MESSAGE	`Misformed octal number`
	CAUSE	An octal number starting with 0 contains an 8 or 9.
	ACTION	Re-enter the command with the correct octal number.
UE448	MESSAGE	`Character constant is missing ending '`
	CAUSE	Token parsed as a character constant is missing a trailing single quotation mark ('). This applies to a single quotation mark followed by a single character or an equivalent backslash sequence.
	ACTION	Re-enter the command enclosing the character constant in single quotation marks (').
UE449	MESSAGE	`String constant is missing ending "`
	CAUSE	Token parsed as a string constant was missing a trailing double quotation mark before the end of the command line.
	ACTION	Re-enter the string with a beginning and ending double quotation marks.
UE450	MESSAGE	`Macros nested too deeply`
	CAUSE	A user specified macro has caused the evaluation of over 20 macro definitions during its evaluation. The debugger cannot evaluate macros nested this deep. This error can also be caused by a recursive macro definition.
	ACTION	Redefine the macro using fewer than 20 macro definitions, or remove the recursive definition.
UE451	MESSAGE	`Macros processing overflow`
	CAUSE	While evaluating a user specified macro, the buffer used to hold the resulting definition for this macro was about to overflow, and the processing for this macro terminated unsuccessfully.
	ACTION	Undefine the unnecessary macros and redefine the macro.

UE452	MESSAGE	`Sorry, you can't access a naked field`
	CAUSE	An attempt was made to refer to a field by name without specifying the qualifying structure (for example, union, record, pointer, etc.).
	ACTION	Use the `\t` display format on the structure object to examine its type information.
UE453	MESSAGE	`Too many subscripts`
	CAUSE	An attempt was made to dereference an array with more dimensions than it was declared to have. However, HP C does allow you to dereference pointers in this manner.
	ACTION	Use the `\t` display format for on the array object to examine its type information.
UE455	MESSAGE	`Invalid field access: "NAME"`
	CAUSE	An attempt was made to do a field dereference of an object (NAME) that was not a structure or union.
	ACTION	Use the `\t` display format to determine the characteristics of the object (NAME).
UE456	MESSAGE	`No such field name "NAME" for that record`
	CAUSE	The record did not contain a field of that NAME.
	ACTION	Use the `\t` display format for more information.
UE457	MESSAGE	`No such field name "NAME" for that struct`
	CAUSE	The `struct` did not contain a field of that NAME.
	ACTION	Use the `\t` display format for more information.

UE458	MESSAGE	`No such field name "NAME" for that union`
	CAUSE	The `union` did not contain a field of that NAME.
	ACTION	Use the `\t` display format for more information.
UE459	MESSAGE	`Illegal cast`
	CAUSE	The expression contains an illegal cast.
	ACTION	Re-enter the command with a valid expression. When casting with a class, structure, or union type, the keyword `class`, `struct`, or `union` must be given.
UE461	MESSAGE	`No child process or corefile`
	CAUSE	The debugger attempted an operation that required an active child process or a core file.
	ACTION	Start a child process using any of the `r` (`run`) or `s` (`step`) commands, or restart the debugger on a valid core file.
UE464	MESSAGE	`Operand stack overflow`
	CAUSE	An expression was too complicated for the expression handler to parse. A combination of more than 15 nested parentheses and/or pending operators may be the cause.
	ACTION	Re-enter the expression, using less than 15 nested parentheses.
UE465	MESSAGE	`Can't execute child program`
	CAUSE	The debugger could not execute the object file given.
	ACTION	Check to see that the file is executable and writable by the user.
UE466	MESSAGE	`Window mode required for this command`
	CAUSE	The debugger was probably invoked with the `-L` option.
	ACTION	Verify that you are using an HP terminal and rerun the debugger without the `-L` option.

UE475	MESSAGE	`Count must be positive`
	CAUSE	The count argument given to the `c` (`continue`) command is negative or 0.
	ACTION	Re-enter the command with a positive count (or none).
UE476	MESSAGE	`Too many characters in wide-character constant`
	CAUSE	More than one valid (possibly multi-byte) character was entered.
	ACTION	Re-enter the expression with one character constant.
UE477	MESSAGE	`Wide string constant too long; truncating to NUM wide-characters`
	CAUSE	Not enough buffer space was available in the user process to store the entire string constant (maximum is 127).
	ACTION	Enter a shorter string constant.
UE479	MESSAGE	`Empty hex escape sequence`
	CAUSE	An invalid ANSI C hexadecimal escape sequence was entered.
	ACTION	Replace the invalid escape sequence with a valid one of the form `\xhh`.
UE480	MESSAGE	`Long double function calls are not supported`
	CAUSE	There was an attempt to call from the command line a function whose return type is long double.
	ACTION	This is not supported.
UE481	MESSAGE	`Long double parameters are not supported in command line function calls`
	CAUSE	There was an attempt to call from the command line a function which expects a long double parameter.
	ACTION	This is not supported.

UE482	MESSAGE	`Unknown print-mode`
	CAUSE	There was an attempt to assign an illegal value to the `$print` debugger variable.
	ACTION	Assign one of these values: `ASCII`, `native`, `raw`.
UE483	MESSAGE	`Misformed binary number`
	CAUSE	A misformed binary number was found in an expression.
	ACTION	Replace the misformed number with a valid one. (`0b` or `0B` followed by one or more 0's or 1's)
UE484	MESSAGE	`Can't open "FILE" as state file`
	CAUSE	The file already exists and is not writable, or the directory has the wrong permissions.
	ACTION	Remove the old file, or make the directory writable and executable.
UE486	MESSAGE	`Can't open "FILE" as restore file`
	CAUSE	The file doesn't exist or the directory is not readable.
	ACTION	Enter a valid file name or add read permission to the directory.
UE488	MESSAGE	`No restore name specified`
	CAUSE	No file name was specified with the `-R` option.
	ACTION	Invoke the debugger with a restore file name or don't provide the `-R` option.
UE490	MESSAGE	`Wrong objectfile for this statefile`
	CAUSE	The save file specified was not created with the object file you are trying to debug.
	ACTION	Specify a valid state file, or if you must use the one originally specified, start the debugger and use the file as a playback file. Be sure to read the warnings related to state files before doing this.

UE605	MESSAGE	`Incompatible debug information`
	CAUSE	The debugger was invoked on a file linked on a older version of the operating system.
	ACTION	Try relinking your program. If that doesn't solve the problem, you will have to recompile the program.
UE626	MESSAGE	`Attempt to read from ODD address`
	CAUSE	An attempt to read from a non-word or half-world boundary was made.
	ACTION	Do not try to read from an odd address. Note: Memory accesses are done word-at-a-time, regardless of how data is formatted in memory.
UE632	MESSAGE	`Wide-character constant not allowed ($lang must be 'C')`
	CAUSE	Attempt to use a wide character constant while the language is not C.
	ACTION	Set `$lang` to `C` and re-enter the expression.
UE642	MESSAGE	`No child process AND no corefile registers`
	CAUSE	The debugger attempted an operation that required an active child process or a core file.
	ACTION	Start a child process using any of the `r` (`run`) or `s` (`step`) commands, or restart the debugger on a valid core file.
UE644	MESSAGE	`- registers bad in core file`
	CAUSE	Unexpected register save area size. The core file might be corrupted.
	ACTION	Create a new core file.
UE645	MESSAGE	`- exec area bad in core file`
	CAUSE	Unexpected exec area size. The core file might be corrupted.
	ACTION	Create a new core file.

UE654	MESSAGE	`Breakpoint count ignored`
	CAUSE	A count is meaningless for class, overload, or instance breakpoints on multiple member functions.
	ACTION	None required. The count was ignored but the breakpoint was set.
UE655	MESSAGE	`This does not appear to be a struct or union`
	CAUSE	The `S` display format was specified but the type of the object to print is not a `struct` or `union`.
	ACTION	If you want to do a formatted dump of an address, cast the address to some `struct` or `union`.
UE659	MESSAGE	`No functions`
	CAUSE	There are no functions to list starting with the provided prefix.
	ACTION	Re-enter the command with a valid function prefix, or just use the `lp` (`list procedure`) command with no prefix to see a list of all the functions.
UE661	MESSAGE	`Cannot view (no debug information for file)`
	CAUSE	A location was specified as *file*:*procedure* and the file is not in the debugger's list of files for which it has debugging information.
	ACTION	Re-enter the command with a valid file name.
UE662	MESSAGE	`Cannot set breakpoint (no debug information for file)`
	CAUSE	The breakpoint location was specified as *file*:*procedure* and the file is not in the debugger's list of files for which it has debugging information.
	ACTION	Re-enter the command with a valid file name. Use the `lf` (`list files`) command to list all valid source files and the path name you must use.

UE663	MESSAGE	`Invalid file on "breakpoint" command`
	CAUSE	A file specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the `lf` (`list files`) command to list all valid source files and the path name you must use.
UE664	MESSAGE	`Invalid procedure on "breakpoint" command`
	CAUSE	A procedure specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the `lp` (`list procedures`) command to see a list of all valid procedures.
UE665	MESSAGE	`Invalid label on "breakpoint" command`
	CAUSE	A label specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE666	MESSAGE	`Invalid class on "breakpoint" command`
	CAUSE	A class specified as part of a breakpoint location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.
UE669	MESSAGE	`Invalid file on "continue" command`
	CAUSE	A file specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the `lf` (`list files`) command to list all valid source files and the path name you must use.

UE670	MESSAGE	`Invalid procedure on "continue" command`
	CAUSE	A procedure specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the `lp` (`list procedures`) command to see a list of all valid procedures.
UE671	MESSAGE	`Invalid line number on "continue" command`
	CAUSE	A line specified as part of a continue location is out of range for the associated file.
	ACTION	Re-enter the command with a valid line number.
UE672	MESSAGE	`Invalid label on "continue" command`
	CAUSE	A label specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE673	MESSAGE	`Invalid class on "continue" command`
	CAUSE	A class specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.
UE676	MESSAGE	`Invalid file on "Continue" command`
	CAUSE	A file specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the `lf` (`list files`) command to list all valid source files and the path name you must use.

UE677	MESSAGE	`Invalid procedure on "Continue" command`
	CAUSE	A procedure specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the `lp` (`list procedures`) command to see a list of all valid procedures.
UE678	MESSAGE	`Invalid line number on "Continue" command`
	CAUSE	A line specified as part of a continue location is out of range for the associated file.
	ACTION	Re-enter the command with a valid line number.
UE679	MESSAGE	`Invalid label on "Continue" command`
	CAUSE	A label specified as part of a continue location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE683	MESSAGE	`Invalid file on "view" command`
	CAUSE	A file specified as part of a view location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the `lf` (`list files`) command to list all valid source files and the path name you must use.
UE684	MESSAGE	`Invalid procedure on "view" command`
	CAUSE	A procedure specified as part of a view location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the `lp` (`list procedures`) command to see a list of all valid procedures.

UE685	MESSAGE	`Invalid line number on "view" command`
	CAUSE	A line specified as part of a view location is out of the range of the associated file.
	ACTION	Re-enter the command with a valid line.
UE686	MESSAGE	`Invalid label on "view" command`
	CAUSE	A label specified as part of a view location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE690	MESSAGE	`Invalid file on CMD command`
	CAUSE	A file specified as part of a CMD location is not known to the debugger.
	ACTION	Re-enter the command with a valid file name. Use the `lf` (`list files`) command to list all valid source files and the path name you must use.
UE691	MESSAGE	`Invalid procedure on CMD command`
	CAUSE	A procedure specified as part of a CMD location is not known to the debugger.
	ACTION	Re-enter the command with a valid procedure name. Use the `lp` (`list procedures`) command to see a list of all valid procedures.
UE692	MESSAGE	`Invalid label on CMD command`
	CAUSE	A label specified as part of a CMD location is not known to the debugger.
	ACTION	Re-enter the command with a valid label name.
UE693	MESSAGE	`Invalid class on CMD command`
	CAUSE	A class specified as part of a CMD location is not known to the debugger.
	ACTION	Re-enter the command with a valid class name.

UE696	MESSAGE	`Must specify breakpoint to delete`
	CAUSE	Although there is a breakpoint at the current viewing location, a breakpoint number must be given with the `db` (`delete breakpoint`) command.
	ACTION	Use the `lb` (`list breakpoints`) command to find the number of the breakpoint you want to delete and re-enter the `db` (`delete breakpoint`) command with the breakpoint number.
UE697	MESSAGE	`Must specify function name`
	CAUSE	The `bpo` (`breakpoint overload`) command was invoked without a function name.
	ACTION	Re-enter the command with a function name.
UE698	MESSAGE	`Function not found`
	CAUSE	No function matching the function name argument given to the `bpo` (`breakpoint overload`) command was found.
	ACTION	Re-enter the command with a valid function name. Use the `lp` (`list procedures`) command to see a list of all valid procedures.
UE709	MESSAGE	`Must specify breakpoint to suspend`
	CAUSE	Although there is a breakpoint at the current viewing location, a breakpoint number must be given with the `sb` (`suspend breakpoint`) command.
	ACTION	Use the `lb` (`list breakpoints`) command to find the number of breakpoints you want to suspend and re-enter the `sb` (`suspend breakpoint`) command with the breakpoint number.

UE710	MESSAGE	`Must specify breakpoint to activate`
	CAUSE	Although there is a breakpoint at the current viewing location, a breakpoint number must be given with the `ab` (`activate breakpoint`) command.
	ACTION	Use the `lb` (`list breakpoints`) command to find the number of breakpoints you want to activate and re-enter the `ab` (`activate breakpoint`) command with the breakpoint number.
UE726	MESSAGE	`Line not found in body of procedure`
	CAUSE	There was an attempt to get the address of a *line* using the notation *function*`#`*line* where *line* is not in the body of the function.
	ACTION	Re-enter the expression with a valid function/line number combination. Use the `lp` (`list procedures`) command with the procedure's name. The range of valid line numbers will be displayed with the procedure.
UE729	MESSAGE	`Invalid structure access`
	CAUSE	There was an attempt to use a non-pointer or a pointer to a class member as a pointer, that is, `p->i` where `p` is not of type pointer.
	ACTION	Re-enter the expression with a valid pointer, or use the address of `p` if you need it, that is, `&(p)->i`
UE731	MESSAGE	`Cannot assign to function`
	CAUSE	There was an attempt to assign a value to a function.
	ACTION	This is not supported by the debugger.
UE732	MESSAGE	`Nil character constant`
	CAUSE	There was an attempt to use `''` as a character.
	ACTION	Re-enter the expression with a valid character constant. `'c'`, or `'\`*value*`'`

UE733	MESSAGE	`Invalid procedure given for "breakpoint trace" command`
	CAUSE	The debugger could not find a procedure with the specified name.
	ACTION	Use the `lp` (`list procedures`) command to find what procedures are known to the debugger, and re-enter the command with the corrected name. Alternatively, if the procedure you supplied was not compiled with the debug flag, you can still set a breakpoint at its entry point by using the '`ba` *address*' command.
UE734	MESSAGE	`Invalid procedure given for "bt" command`
	CAUSE	The debugger could not find a procedure with the specified name.
	ACTION	Use the `lp` (`list procedures`) command to find what procedures are known to the debugger, and re-enter the command with the corrected name. Alternatively, if the procedure you supplied was not compiled with the debug flag, you can still set a breakpoint at its entry point by using the '`ba` *address*' command.
UE756	MESSAGE	`No registers in core file -- registers required`
	CAUSE	The core file has a format not recognized by the debugger.
	ACTION	Obtain a new core file on the same system as the debugger you are running.
UE757	MESSAGE	`Modifier is not allowed before CMD command`
	CAUSE	In `cdb`, `fdb`, or `pdb`, a modifier was entered before a command that does not take a modifier.
	ACTION	Re-enter the command without a modifier in front of it.

UE785	MESSAGE	`Address is required after "va"`
	CAUSE	The `va` command was entered with no parameter.
	ACTION	Re-enter the command with an address argument.

Debugger Errors (DB1-DB8)

DB1	MESSAGE	`Assigning to NUM byte object from NUM byte object; moved NUM bytes`
	CAUSE	The object on the left side of an assignment was not equal to the size of the right side of the expression. The debugger copied a series of bytes equal in size to the left side of the assignment statement.
	ACTION	Re-enter the command, using expressions of equal length, or else results based on truncation will occur.
DB10	MESSAGE	`WARNING: TOO FEW PARAMETERS`
	CAUSE	An attempt was made to call a debuggable procedure from the command line with a different number of parameters than specified in the symbol table. The procedure can still be called, but it may lead to odd results which depend on the language and the called procedure.
	ACTION	Use the *V* (*View*) command to view the procedure to determine the correct number of parameters.
DB11	MESSAGE	`WARNING: TOO MANY PARAMETERS`
	CAUSE	An attempt was made to call a debuggable procedure from the command line with a different number of parameters than specified in the symbol table. The procedure can still be called, but it may lead to odd results which depend on the language and the called procedure.
	ACTION	Use the *V* (*View*) command to view the procedure to determine the correct number of parameters.

B

HP C Language Operators

This appendix lists and describes operators for the HP C programming language that the debugger expression evaluator recognizes.

HP C Language Operators

The following table lists the supported HP C operators. Operators are listed in order of precedence, from highest to lowest. All operators listed in the same box are of equal precedence. Associativity of operators in the following table is from left to right, unless otherwise stated.

For HP C, the operators && and || are not short circuited as is done by the HP C compiler; all portions of an expression involving these operators are evaluated. Also, HP C pointer arithmetic in the debugger is unsupported.

Full support of *struct* objects is provided.

HP C Language Operators

Table B-1. Language Operators for HP C

Operator	Operation
( )	parenthesis (group elements)
[ ]	array member selection
->	member selection of pointer to structure
.	member selection of structure
! (*order is right to left*)	unary logical negation
˜ (*order is right to left*)	unary logical one's complement
- (*order is right to left*)	unary negation
* (*order is right to left*)	unary indirection (pointer or address dereferencing)
& (*order is right to left*)	unary address of an object
sizeof (*order is right to left*)	unary size of an object
*	multiplication
/	division
%	modulus - mod function
+	addition
-	subtraction
<<	bit-wise logical left shift; fill with 0
>>	bit-wise arithmetic right shift; unsigned fill with 0, else fill with sign bit
<	relational less than
<=	relational less than or equal to
>	relational greater than
>=	relational greater than or equal to
==	relational equal to
!=	relational not equal to

Table B-1. Language Operators for HP C (continued)

Operator	**Operation**
&	bit-wise logical and
^	bit-wise logical exclusive or
|	bit-wise logical inclusive or
&&	logical and
||	logical or
=(*order is right to left*)	assignment
op=(*order is right to left*)	assignment operators of the form: e1 op= e2 which means (e1) = (e1) op (e2). Op may be any one of the mathematical or bit-wise operators (*, /, %, +, <<, >>, &, ^, |)
Special operators:	
$addr	unary address of an object
$sizeof	unary size of an object
$in	unary suspended in named routine

C

HP FORTRAN 77 Language Operators and VMS Record Support

This appendix lists and describes operators for the HP FORTRAN 77 programming language that the debugger expression evaluator recognizes.

HP FORTRAN 77 Language Operators

The following table lists the supported HP FORTRAN 77 operators. Operators are listed in order of precedence, from highest to lowest. All operators listed in the same box are of equal precedence. All operators of equal precedence evaluate left to right, except assignment. Assignment is treated by the debugger as an operator.

Associativity of operators in the following table is from left to right, unless otherwise stated.

Complex variables in HP FORTRAN 77 are not supported except as a pair of two separate reals or doubles. Any HP C language operators that do not clash with supported HP FORTRAN 77 operators can be used in HP FORTRAN 77 expressions, with the corresponding C interpretation. The only exception to this is the unary operator *sizeof*.

HP FORTRAN 77 Language Operators

Table C-1. Language Operators for HP FORTRAN 77

Operator	Operation
( )	parentheses (grouping), array member selection
*	multiplication
/	division
+	addition
-	subtraction or unary negation
.LT.	relational less than
.LE.	relational less than or equal to
.EQ.	relational equal to
.GE.	relational greater than or equal to
.NE.	relational not equal to
.GT.	relational greater than
.NOT.	logical negation
.AND.	logical and
.OR.	logical or
.EQV.	logical equivalence
.NEQV.	logical nonequivalence
= (*order is right to left*)	assignment
Special operators:	
$addr	unary address of an object
$sizeof	unary size of an object
$in	unary suspended in named routine

VMS FORTRAN Records

HP Symbolic Debugger provides support for VMS FORTRAN records. There are four associated types:

- structures
- records
- unions
- maps

A *structure* defines record field types such as in the following example:

```
structure /date/
 integer a
 union
     map
      integer b
      real c
      character*8 d
      integer e
      union
          map
           logical f
           integer g
          end map
          map
           character*3 h
          end map
          map
           real i
          end map
      end union
     end map
 end union
 real j
 integer f
end structure
```

A *record* corresponds to an instance of that record structure.

For example, given the previous structure, you can now define a record with that structure:

```
record /date/ rec1
```

In HP Symbolic Debugger, HP FORTRAN 77 *records* are treated as HP FORTRAN 77 *structures* from the debugger. This means that if you use the *print* command with the \t format to look at a record, you will see the record's *structure* rather than the record definition, `record /date/ rec1`.

For example, if you type:

```
>p rec1\t
```

you will get:

```
structure /date/
 integer a
 union
     map
      integer b
      real c
      character*8 d
      integer e
      union
          map
           logical f
           integer g
          end map
          map
           character*3 h
          end map
          map
           real i
          end map
      end union
     end map
 end union
 real j
 integer f
end structure rec1
```

You can access any element within a record. Because maps and unions are unnamed, they are ignored in naming subelements. For example, field `h` in the previous example must be accessed as:

```
rec1.h
```

If there is any ambiguity among field names, the first one appearing by a given name is chosen, just as it is in HP FORTRAN 77. For example, field `rec1.f` in the example above is of type *logical*, not integer.

When the value or type of any field in a record is displayed, its individual format is identical to what it would be if it were not within a record. For the records, unions, and maps themselves, these keywords are used identically to the way they are used in HP FORTRAN 77 except:

- When printing the type of a structure, its name will follow the entire structure instead of preceding it.

  For example:

  ```
  >p rec\t
  ```

  gives you this:

  ```
  structure
   integer*4 i
  end structure rec
  ```

- When printing the value of a structure, its name and an equal sign (=) precede its value.

  For example:

  ```
  >p rec
  ```

  gives you this:

  ```
  rec = structure
   i = 3
  end structure
  ```

D

HP Pascal Language Operators

This appendix lists and describes operators for the HP Pascal programming language that the debugger expression evaluator recognizes.

HP Pascal Language Operators

The following table lists the supported HP Pascal operators. Operators are listed in order of precedence, from highest to lowest. All operators listed in the same box are of equal precedence. All operators of equal precedence evaluate left to right, except assignment. Assignment is treated by the debugger as an operator.

Associativity of operators in the following table is from left to right, unless otherwise stated.

Any HP C language operators that do not clash with supported HP Pascal operators can be used in HP Pascal expressions, with the corresponding C interpretation.

There are two restrictions with the language operators for HP Pascal:

- Variables qualified by the *WITH* statement in an HP Pascal program must be fully qualified in HP Symbolic Debugger expressions. The HP Pascal *WITH* construct is not recognized as a debugger command.
- The debugger does not support HP Pascal set constants and does not support operations on sets.

HP Pascal Language Operators

Table D-1. Language Operators for HP Pascal

Operator	Operation
( )	parenthesis, group elements
[ ]	array member selection
.	member selection of record
^ (*order is right to left*)	pointer (address) dereferencing
NOT (*order is right to left*)	unary logical negation
sizeof (*order is right to left*)	unary size of an object
*	multiplication
/	real division
DIV	integer division with truncation
MOD	modulus
+	addition
-	subtraction
<	relational less than
>	relational greater than
<=	relational less than or equal to
>=	relational greater than or equal to
=	relational equal to
< >	relational not equal to
:= (*order is right to left*)	assignment
AND	logical and
OR	logical or
Special operators:	
$addr	unary address of an object
$sizeof	unary size of an object
$in	unary suspended in named routine

E

HP COBOL II Language Operators

This appendix lists and describes operators for the HP COBOL II programming language that the debugger expression evaluator recognizes.

HP COBOL II Language Operators

The following table lists the supported HP COBOL II operators. Operators are listed in order of precedence, from highest to lowest. All operators listed in the same box are of equal precedence. All operators of equal precedence evaluate left to right, except assignment.

Associativity of operators in the following table is from left to right, unless otherwise stated.

Table E-1. Language Operators for HP COBOL II

Operator	Operation
( )	parenthesis, group elements
*	multiplication
/	division
+	addition
-	subtraction
<	relational less than
>	relational greater than
<=	relational less than or equal to
>=	relational greater than or equal to
=	relational equal to
< >	relational not equal to
AND	logical and
OR	logical or
NOT	unary logical negation
`move`	assignment

Dereferencing Operations

There are two supported HP COBOL II dereferencing operations, *field dereferencing* (*variable qualification*) and *array dereferencing*.

Field Dereferencing

There are two operators that are supported for field dereferencing (variable qualification):

- .
- of

The difference in these operators is the order in which the fields are listed. The . (*dot*) operator is used to specify a qualified path from the parent field down; the *of* operator is used to specify the path from the child field up. For example, in the pseudo-COBOL structure:

```
01 fob
   02 bar
      03 stooge
         04 curly
         04 moe
         04 larry
   02 bat
      03 marx
         04 harpo
         04 chico
         04 groucho
```

the fully qualified path to print *chico* would be either of the two commands listed below.

```
disp fob.bat.marx.chico
```

OR

```
disp chico of marx of bat of fob
```

It is not always necessary to fully qualify a field; the minimum list of parent fields that uniquely identify the field will suffice. In other words, if there is only one field named "chico" in all the variables in the current subprogram, then `disp chico` is sufficient. Suppose, however, you had the following structure.

```
01 fob
   02 bar
      03 marx
         04 harpo
         04 chico
         04 groucho
   02 bat
      03 marx
         04 harpo
         04 chico
         04 groucho
```

With this structure, `chico` would not be unique, nor would `marx.chico`. The minimum qualification necessary is either:

```
disp bat.chico
```

OR

```
disp chico of bat
```

If a name is fully qualified such as `fob.bat.marx.chico`, it must always be unique.

Array Dereferencing

HP COBOL subscripts applied to a structure are always listed at the *end* of the field list. For example, using this structure:

```
01 fob
   02 bar
      03 stooge
         04 curly
         04 moe
         04 larry
   02 bat
      03 marx
         04 harpo
         04 chico
         04 groucho
```

If *fob* and *marx* were tables, a valid expression might be:

```
fob.bat.marx.chico(3,7)
```

The debugger determines to which fields the subscripts apply. The more conventional form:

```
fob(3).bat.marx(7).chico
```

is *NOT* legal.

F

Special Variables Used by the Symbolic Debugger

This appendix covers special variables that are not normally directly accessible.

Special Variables

Table F-1. Special Variables

Variable	Description
`$`*var*	Represents user-defined variables. They are of type long integer and do not take on the type of any expressions assigned to them.
Hardware Registers	Represents the names of the registers, the program counter and stack, data, argument and return-value pointers.
`$r0` ... `$r31`	General Registers
`$f0` ... `$f15`	Floating Point Registers
`$pc`	Program Counter
`$sp`	Stack Pointer
`$dp`	Data Pointer
`$arg0` ... `$arg3`	Argument Registers
`$ret0` ... `$ret1`	Return-value Registers
`$result`	References the return value from the last command line procedure call. $*short* and $*long* are used as other ways of viewing $*result*. Where possible, $*result* takes on the type of the procedure.
`$lang`	Allows you to view and modify the current source language flag for expression evaluations. Valid values for $lang are *C*, *FORTRAN*, *Pascal*, *COBOL*, and *default*. When $*lang* is set to "default", any language expression syntax is always the same as the source language of the procedure currently being viewed.

Table F-1. Special Variables (continued)

Variable	Description
`$line`	Displays the current source line number (the next statement to be executed). It is automatically set by a number of different commands.
`$malloc`	Allows you to see the amount of memory (in bytes) currently allocated by the debugger for its own use. This does not reflect memory use of the program being debugged.
`$step`	Allows you to see and change the number of machine instructions the debugger steps through while in a non-debuggable procedure, before setting an uplevel breakpoint and free-running to it. This situation occurs only when the program is executing in single-step or assertion mode.

G

Limitations and Hints

This appendix lists some limitations of HP Symbolic Debugger and gives some hints for debugger usage.

Limitations and Hints

- CNTRL Y should be a trap that performs like the hardware traps. That is, if the child process is running, it should be forced to stop with control transferred to the debugger. This allows infinite loops to be temporarily broken without aborting the debugger. However, the MPE/iX operating system currently handles CNTRL Y as a special case and unless the child explicitly requests its own handler, the CNTRL Y trap will only be detected by the debugger when it occurs during execution of the debugger's own code.
- Do not modify any file while the debugger has it open. If you do, the debugger gets confused and might display garbage.
- Some statements do not emit code where you would expect it. For example, assume:

  ```
   99:   for (i=0; i<9; i++) {
  100:          xyz (i);
  101:   }
  ```

  A breakpoint placed on line 99 will be hit only once in some cases. The code for incrementing is placed at line 101. Each compiler is a little different; you must get used to what your particular compiler does. A good way of finding out is to use single stepping to see in what order the source lines are executed.
- Some compilers only issue source line symbols at the end of each logical statement or physical line, *whichever is greater*. This means that, if you are in the habit of saying `a=0; b=1;` on one line, there is no way to put a breakpoint after the assignment to `a` but before the assignment to `b`.
- The debugger does not support identically-named procedures, except in HP Pascal if the procedures are in different scopes. The debugger will always use the first procedure with the given name.
- There is no support for HP Pascal packed arrays where the element size is not a whole number of bytes. Any reference into such an array might produce garbage or a bad access.
- Assignments into objects greater than four bytes in size from debugger special variables, result in errors or invalid results.

- The debugger supports call-by-reference only for known parameters of known (debuggable) procedures. If the object to pass lives in the child process, you can fake such a call by passing &*object*, for example, the address of the object.
- Only the first number of a complex pair is passed as a parameter. Functions which return complex numbers are not called correctly; insufficient stack space is allocated for the return area, which can lead to overwriting the parameter values.

H

Installed Files

This appendix lists the installed files for the HP Symbolic Debugger.

Debugger Installation

These are the files needed to use the HP Symbolic Debugger on your system.

- *xdbend.lib.sys*

  The file *xdbend.lib.sys* must be linked at the end of the user program to give the debugger private data space in the user process.

- *xdb.pub.sys*

  The file *xdb.pub.sys* is the HP Symbolic Debugger/iX executable program file with shared access.

- *pxdb.pub.sys*

  The file *pxdb.pub.sys* (the preprocessor) processes the executable file the first time the debugger is invoked on it. It produces quick-lookup tables to increase the performance of the debugger and removes duplicate global definitions.

- *xdbhelp.pub.sys*

  The file *xdbhelp.pub.sys* contains the text of the *Help* facility, which is a summary of HP Symbolic Debugger commands.

- *xdbcat.lib.sys*

  The file *xdbcat.lib.sys* contains the message catalog.

I

HP Symbolic Debugger Commands

This section describes command syntax and gives a description of all the HP Symbolic Debugger commands.

Enter the following command to start the debugger:

```
                   [          [ -d group[ .acct] ]               ]
                   [          [ -r file          ]               ]
run xdb.pub.sys    [ ;info="  [ -p file          ]  [ ... ]"     ]
                   [          [ -L               ]               ]
                   [          [ -S num           ]               ]
                   [          [ objectfile       ]               ]
```

The HP Symbolic Debugger options are described below:

`-d` *group.[acct]*	This option names an alternate group and (optional) account containing the source files used to create the *objectfile*. Group and accounts are searched in the order that you list them. The current group and account is used if the file is not found in the group and account that you enter here. You can enter more than one `-d` option.
`-p` *file*	This option names a playback file created in a previous debugger session (see the `-r` option) or one that you created yourself.
`-r` *file*	This option names the file to which all debugger commands that you enter are recorded. You can use this file as a playback file in subsequent debug sessions (see the `-p` parameter). Recording begins as soon as you start the debugger. Any previous contents of the file are overwritten (no appending takes place).
`-L`	This option allows you to use the debugger in line mode when you do not have a terminal that supports memory lock.
`-S` *num*	This option sets the string-cache size to the number of bytes specified. The string cache holds data read from the *objectfile*. The default is 1024 bytes (1kb). Increasing the string cache size can improve debugger performance for large programs.
objectfile	This argument names the file that contains the executable code for the program. If you do not enter this option, you will be prompted for the *objectfile*. If this is the first time you are running the debugger, *objectfile* will be preprocessed to allow faster debugger startups in subsequent sessions.

Window Mode Commands

Table I-1. Window Mode Commands

Cmd	Syntax	Description
`fr`	$\left\{\begin{array}{l}\texttt{fr}\\\texttt{floating point registers}\end{array}\right\}$	Displays the PA-RISC floating point registers in the register window when the debugger is in disassembly mode. Each register appears as a two word pair (two sets of eight hexadecimal digits).
`gr`	$\left\{\begin{array}{l}\texttt{gr}\\\texttt{general registers}\end{array}\right\}$	Displays the PA-RISC general registers in the register window when the debugger is in disassembly mode.
`sr`	$\left\{\begin{array}{l}\texttt{sr}\\\texttt{special registers}\end{array}\right\}$	Displays the PA-RISC special registers (space and control) when the debugger is in disassembly mode.
`td`	$\left\{\begin{array}{l}\texttt{td}\\\texttt{toggle disassembly}\end{array}\right\}$	Toggles the source window between disassembly mode and source mode.
`ts`	$\left\{\begin{array}{l}\texttt{ts}\\\texttt{toggle screen}\end{array}\right\}$	Toggles the source window between all source or all assembly and split screen mode.
`u`	$\left\{\begin{array}{l}\texttt{u}\\\texttt{update}\end{array}\right\}$	Updates the source and location windows to show the current location of the user program.
`U`	$\left\{\begin{array}{l}\texttt{U}\\\texttt{Update}\end{array}\right\}$	Clears the screen of data and redraws the screen.
`w`	$\left\{\begin{array}{l}\texttt{w}\\\texttt{window}\end{array}\right\}$ *number*	If your terminal supports windowing, this command changes the size of the source window to the number of lines that you specify. Enter a number from 1 to 21.

File Viewing Commands

Table I-2. File Viewing Commands

Cmd	Syntax	Description
`+`	`+` [*number*]	Moves forward in the current file the specified number of lines (or the specified number of instructions in disassembly mode). If you do not enter a number, the next line (or instruction) becomes the current line (or instruction).
`-`	`-` [*number*]	Moves the specified number of lines (or the specified number of instructions in disassembly mode) backward in the current file and updates the windows. The default is one line (or instruction) before the current line (or instruction).
`/`	`/` [*string*]	Searches forward in the file for the specified string. Searches wrap around the end of the file. If you do not enter a string, the last one that you entered is used again. The string must be literal; wild cards are not supported.
`?`	`?` [*string*]	Searches backward in the current file for a specific pattern. Searches wrap around the beginning of the file. If you do not enter a string, the last search string is used again. The string must be literal; wild cards are not supported.
`D`	{`D` | `Directory`} "*dir*"	Adds the directory that you specify to the list of directory search paths for source files.
`ld`	{`ld` | `list directories`}	Lists all the alternate directories that are searched when the debugger tries to locate the source files.
`lf`	{`lf` | `list files`} [*string*]	Lists all source files containing executable statements that were compiled to build the executable file. If a string is specified, only those files beginning with the string are listed.
`L`	{`L` | `Location`}	Displays in the command window the current file, procedure, line number and the source text for the current point of execution.
`n`	{`n` | `next`}	Repeats the previous search (/ or ?) command.
`N`	{`N` | `Next`}	Repeats the previous search (/ or ?) command, searching in the opposite direction.

Table I-2. File Viewing Commands (continued)

Cmd	Syntax	Description
`v`	$\left\{\begin{array}{l}\texttt{v}\\ \texttt{view}\end{array}\right\}\left[\,location\,\right]$	Displays one source window forward from the current source window. One line from the previous window is preserved for context. If your terminal does not support windowing, only the new source line is displayed. Using the *location* option causes the specified location to become the current location, and the source at the specified location is then displayed in the source window.
`V`	$\left\{\begin{array}{l}\texttt{V}\\ \texttt{View}\end{array}\right\}\left[\,depth\,\right]$	Displays the text for the procedure at the depth on the program stack that you specify. If you do not enter a depth, the current active procedure is used.
`va`	$\left\{\begin{array}{l}\texttt{va}\\ \texttt{view address}\end{array}\right\}$ *address*	Displays in the source window assembly code at the specified address. A specified address can be an absolute address or symbolic code label with an optional offset (for example, _start + 0x20).

Data Viewing and Modification Commands

Table I-3. Data Viewing and Modification Commands

Cmd	Syntax	Description
`disp`	$\left\{\begin{array}{l}\texttt{disp}\\ \texttt{display}\end{array}\right\} \textit{var}$	Used only with HP COBOL II programs to print COBOL variables. Simple items, fields, and array elements can be displayed. Items displayed can be of type "edited" or "non-edited".
`l`	$\left\{\begin{array}{l}\texttt{l}\\ \texttt{list}\end{array}\right\}$ $\left[\textit{proc}\ \left[:\textit{depth}\right]\right]$	Lists all parameters and local variables of the current procedure. You can optionally specify any active procedure and its depth on the stack.
`lc`	$\left\{\begin{array}{l}\texttt{lc}\\ \texttt{list common}\end{array}\right\}$ $\left[\textit{string}\right]$	Used when debugging an HP FORTRAN 77 program, this command displays HP FORTRAN 77 common blocks and their associated variables. If a string is specified, only those common blocks whose names begin with that string are printed; otherwise, all common blocks within the current subroutine/function are printed.
`lg`	$\left\{\begin{array}{l}\texttt{lg}\\ \texttt{list globals}\end{array}\right\}$ $\left[\textit{string}\right]$	Lists all global variables and their values. If a string is specified, only those global variables whose names begin with this string are listed.
`ll`	$\left\{\begin{array}{l}\texttt{ll}\\ \texttt{list labels}\end{array}\right\}$ $\left[\textit{string}\right]$	Lists all labels and program entry points known to the linker. If a string is specified, only those symbolic addresses with this prefix are used.
`lm`	$\left\{\begin{array}{l}\texttt{lm}\\ \texttt{list macros}\end{array}\right\}$ $\left[\textit{string}\right]$	Displays all user-defined macros and their definitions. If a string is specified, only those macros whose names begin with this string are listed.
`lp`	$\left\{\begin{array}{l}\texttt{lp}\\ \texttt{list procedures}\end{array}\right\}$ $\left[\textit{string}\right]$	Lists all procedure names and their aliases as well as their locations in memory. If a string is specified, only those procedures whose names begin with this string are listed.
`lr`	$\left\{\begin{array}{l}\texttt{lr}\\ \texttt{list registers}\end{array}\right\}$ $\left[\textit{string}\right]$	Lists all registers and their contents. If a string is specified, only those registers beginning with this string are listed.
`ls`	$\{\ \texttt{ls}$	Lists all special variables and their values. Registers are not listed. If a

Table I-3.
Data Viewing and Modification Commands (continued)

Cmd	Syntax	Description
`mov`	$\left\{\begin{array}{l}\texttt{mov}\\ \texttt{move}\end{array}\right\}$ *expr1* `to` *expr2*	Used only with HP COBOL II programs to modify variables. The first expression is the *source*; the second is the *destination*. The source and destination cannot be an edited field. The source can be any non-edited COBOL field, a string literal, a number, or a named constant (such as SPACES or BLANKS). The destination can be any non-edited COBOL field.
`p`	$\left\{\begin{array}{l}\texttt{p}\\ \texttt{print}\end{array}\right\}$ $\left\{\begin{array}{l} expr\ [?format] \\ \left[\begin{array}{c}+\\ -\end{array}\right] [\,[\backslash]format] \end{array}\right\}$	Displays program data in the formats shown in tables 1-5 and 1-6 of chapter 1, "Reference Tables". A format has the syntax: $[count]\{formchar\}[size]$ *Formchar*, which is required, is the actual format in which you choose to display the data. *Count* is the number of times to apply the format. *Size* is the number of bytes that are formatted for each data item, and overrides the default size for the given format. *p+* prints the next element. *p-* prints the previous element. Use the *\format* option to display the value of the expression in a specific format. Use the *?format* option to print the address of the evaluated expression in the selected format. The *p* (*print*) command is also used to modify the value of a variable when *expr* contains an assignment operator.

Stack Viewing Commands

Table I-4. Stack Viewing Commands

Cmd	Syntax	Description
`t`	$\left\{\begin{array}{l}\texttt{t}\\\texttt{trace}\end{array}\right\}\left[\,depth\,\right]$	Prints a stack trace. You can optionally specify a *depth*. The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed.
`T`	$\left\{\begin{array}{l}\texttt{T}\\\texttt{Trace}\end{array}\right\}\left[\,depth\,\right]$	Prints a stack trace. You can optionally specify a *depth*. The default depth is 20 levels. If an optional depth is supplied, only the procedures up to this depth in the stack are displayed. Displays everything the *t* (*trace*) command displays, plus all local variables and their values in \n format.

Status Viewing Command

Table I-5. Status Viewing Command

Cmd	Syntax	Description
`I`	$\left\{ \begin{array}{l} \texttt{I} \\ \texttt{Inquire} \end{array} \right\}$	Prints the current state of the debugger. The output contains information such as the version number of the debugger, program name, number of source files and procedures, process id of the child process, number of breakpoints, record and playback information, etc.

Job Control Commands

Table I-6. Job Control Commands

Cmd	Syntax	Description
`c`	$\left\{\begin{array}{l}\texttt{c}\\ \texttt{continue}\end{array}\right\}\left[\textit{location}\right]$	Resumes execution after a breakpoint or a signal has been encountered, ignoring the signal, if any. If a location is specified, a temporary breakpoint is set at that location.
`C`	$\left\{\begin{array}{l}\texttt{C}\\ \texttt{Continue}\end{array}\right\}\left[\textit{location}\right]$	Resumes execution after a breakpoint or a signal has been encountered, allowing the signal, if any, to be received by the child process. If a location is specified, a temporary breakpoint is set at that location. This works the same as the *c* (*continue*) command on MPE/iX.
`k`	$\left\{\begin{array}{l}\texttt{k}\\ \texttt{kill}\end{array}\right\}$	Terminates the current child process, if any.
`r`	$\left\{\begin{array}{l}\texttt{r}\\ \texttt{run}\end{array}\right\}$ $\left[\texttt{;info='}\textit{info string}\texttt{'}\right]$	Lets you run a program as a new child process with an optional *info string*. If you do not enter an *info string*, the debugger uses those supplied with the last *r*(*run*) command (if any). *info string* may contain a "<" and/or a ">" for redirecting standard input and standard output (*$STDIN*) and (*$STDOUT*).
`R`	$\left\{\begin{array}{l}\texttt{R}\\ \texttt{Run}\end{array}\right\}$	Lets you run a program as a new child process with no argument list. If a child process already exists, the debugger asks if you want to terminate the child process first. The environment for the child process is the same as that for the debugger.
`s`	$\left\{\begin{array}{l}\texttt{s}\\ \texttt{step}\end{array}\right\}\left[\textit{number}\right]$	Single steps through a program, executing one source statement or machine instruction at a time before pausing and prompting for another command. In source mode, one source statement is executed; in disassembly mode, one machine instruction is executed. If a procedure call is encountered, the procedure is single stepped in the same manner ("stepped into"). To execute more than one statement or instruction, enter that number as the *number* parameter.

Table I-6. Job Control Commands (continued)

Cmd	Syntax	Description
`S`	$\left\{ \begin{array}{l} \texttt{S} \\ \texttt{Step} \end{array} \right\} \left[\textit{number} \right]$	Single steps through a program. In source mode, one source statement (or one step of a multiple step statement in HP Pascal or HP C) is executed; in disassembly mode, one machine instruction is executed (several machine instructions might be equivalent to one source statement). If a procedure call is encountered, it is not "stepped into". Instead, execution steps to the statement following the call. To execute more than one statement or instruction, enter that number as the *number* parameter.

Breakpoint Commands

Overall Breakpoint Commands

Table I-7. Overall Breakpoint Commands

Cmd	Syntax	Description
`lb`	$\left\{\begin{array}{l}\texttt{lb}\\\texttt{list breakpoints}\end{array}\right\}$	Displays all breakpoints in the program, both active and suspended, and the overall breakpoint state.
`tb`	$\left\{\begin{array}{l}\texttt{tb}\\\texttt{toggle breakpoints}\end{array}\right\}$	Toggles the overall breakpoint state from active to suspended or vice versa. The state of the individual breakpoints remains unchanged.

Breakpoint Creation Commands

Table I-8. Breakpoint Creation Commands

Cmd	Syntax	Description
`b`	$\left\{\begin{array}{l}\texttt{b}\\ \texttt{breakpoint}\end{array}\right\}\left[\mathit{location}\right]\left[\backslash \mathit{count}\right]$ $\left[\mathit{command\text{-}list}\right]$	Sets a breakpoint at the location that you specify. If you do not enter a location, the current line in the source window is used. The breakpoint is executed on each occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list.
`ba`	$\left\{\begin{array}{l}\texttt{ba}\\ \texttt{breakpoint address}\end{array}\right\}\mathit{address}\ \left[\backslash \mathit{count}\right]$ $\left[\mathit{command\text{-}list}\right]$	Sets a breakpoint at the specified address. Note that the address can be specified by giving the name of a procedure or an expression containing a name. The breakpoint is executed on each occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list.
`bb`	$\left\{\begin{array}{l}\texttt{bb}\\ \texttt{breakpoint beginning}\end{array}\right\}\left[\mathit{depth}\right]$ $\left[\backslash \mathit{count}\right]\left[\mathit{command\text{-}list}\right]$	Sets a breakpoint at the first executable statement of the procedure at the specified depth on the program stack. If you do not enter a depth, the procedure shown in the source window is used. The breakpoint is executed on each occurrence (count) that you specify.

Table I-8. Breakpoint Creation Commands (continued)

Cmd	Syntax	Description

`bt`	$\left\{\begin{array}{l}\texttt{bt}\\ \texttt{breakpoint trace}\end{array}\right\}\left[\begin{array}{l}\textit{proc}\\ \textit{depth}\end{array}\right]\left[\backslash \textit{count}\right]$ $\left[\textit{command-list}\right]$	Sets a *trace* breakpoint at the current or named procedure or at the procedure that is at the specified depth on the program stack. A breakpoint is set at the entry and exit point of the procedure. If you include a command list, it is executed at the beginning of the procedure or subprogram. The following command list will be executed at the end of the procedure or subprogram.
		
		`{ Q;p $ret0\d;c }`
		
		If you omit a command list, the following two command lists are executed at the beginning and end of the procedure or subprogram, respectively.
		
		`{ Q; t 2; c }`
		`{ Q;p $ret0\d;c }`

Table I-8. Breakpoint Creation Commands (continued)

Cmd	Syntax	Description
`bu`	$\left\{\begin{array}{l}\texttt{bu}\\\texttt{breakpoint uplevel}\end{array}\right\}\left[\,\textit{depth}\,\right]\left[\,\backslash\textit{count}\,\right]$ $\left[\,\textit{command-list}\,\right]$	Sets an *uplevel* breakpoint to occur immediately on return from the procedure at the specified depth on the program stack. If you do not enter a depth, the procedure shown in the source window is used. The breakpoint is executed on each occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list.
`bx`	$\left\{\begin{array}{l}\texttt{bx}\\\texttt{breakpoint exit}\end{array}\right\}\left[\,\textit{depth}\,\right]\left[\,\backslash\textit{count}\,\right]$ $\left[\,\textit{command-list}\,\right]$	Sets an *exit* breakpoint at the epilogue code of the procedure at the specified depth on the program stack. If you do not enter a depth, the procedure shown in the source window is used. The breakpoint is executed on each occurrence (count) that you specify. You can enter a list of commands to be executed at the breakpoint by entering the command list.

Breakpoint Status Commands

Table I-9. Breakpoint Status Commands

Cmd	Syntax	Description
`ab`	$\left\{\begin{array}{l}\texttt{ab}\\ \texttt{activate breakpoint}\end{array}\right\}$ $\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$	Activates the breakpoint having the number (ID) that you specify. If you do not enter a number, the breakpoint at the current line is activated. Use the asterisk (*) to activate all breakpoints, including all-procedure breakpoints.
`bc`	$\left\{\begin{array}{l}\texttt{bc}\\ \texttt{breakpoint count}\end{array}\right\}$ $\mathit{number}\ \mathit{expr}$	Sets the count of the specified breakpoint number to the integer value of the evaluated expression that you enter.
`db`	$\left\{\begin{array}{l}\texttt{db}\\ \texttt{delete breakpoint}\end{array}\right\}$ $\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$	Deletes the breakpoint having the number (ID) that you specify. If you do not enter a number, the breakpoint at the current line is deleted. Use the asterisk (*) to delete all breakpoints including all-procedure breakpoints.
`sb`	$\left\{\begin{array}{l}\texttt{sb}\\ \texttt{suspend breakpoint}\end{array}\right\}$ $\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$	Suspends (deactivates) the breakpoint having the number (ID) that you specify. If you do not enter a number, the breakpoint at the current line is suspended. Use the asterisk (*) to suspend all breakpoints, including all-procedure breakpoints. This also causes the overall breakpoint state to become suspended.

All-Procedures Breakpoint Commands

Table I-10. All-Procedures Breakpoint Commands

Cmd	Syntax	Description
`bp`	{ `bp` `breakpoint procedure` } [*command-list*]	Sets permanent *procedure* breakpoints at the first executable statement of every procedure for which debugger information is available. The breakpoint is encountered each time the procedure is entered. When any entry procedure breakpoint is encountered, the command list is executed.
`bpt`	`bpt` [*command-list*]	Sets permanent *procedure trace* breakpoints at the first and last executable statement of every procedure for which debugger information is available. The breakpoints are encountered each time the procedure is entered. The commands, if any, are associated with the entry breakpoint. If no command list is specified, the entry command list defaults to: `{Q;t 2;c}` The exit command list is: `{Q;p $ret\d;c}`
`bpx`	`bpx` [*command-list*]	Sets permanent *procedure exit* breakpoints after the last executable statement of every procedure for which debugger information is available. The breakpoint is encountered each time the procedure is exited. When any procedure exit breakpoint is encountered, the command list is executed.
`dp`	{ `dp` `delete procedure` }	Deletes all *procedure* breakpoints set with the *bp* (breakpoint procedure) command. All breakpoints set by commands other than the *bp* command will remain in effect.
`Dpt`	`Dpt`	Deletes all *procedure trace* breakpoints at the first and last executable statement of every procedure. All breakpoints set by commands other than the *bpt* command will remain in effect.
`Dpx`	`Dpx`	Deletes all *procedure exit* breakpoints at the last executable statement of every procedure. All breakpoints set by commands other

Global Breakpoint Commands

Table I-11. Global Breakpoint Commands

Cmd	Syntax	Description
`abc`	`abc` *command-list*	Defines a global breakpoint command list which will be executed whenever any user defined breakpoint is encountered. These include normal, procedure, procedure trace, and procedure exit breakpoints.
`dbc`	`dbc`	Deletes the global breakpoint command list.

All-Paragraph Breakpoint Commands

Table I-12. Paragraph Breakpoint Commands

Cmd	Syntax	Description
`bpg`	{ `bpg` / `breakpoint paragraph` } [*command-list*]	Sets permanent *paragraph* breakpoints at the first executable statement of every HP COBOL II paragraph and section for which debugger information is available. The breakpoint is encountered each time the paragraph or section is entered. When any entry paragraph breakpoint is encountered, the *command list* is executed.
`dpg`	{ `dpg` / `delete paragraph` }	Deletes all *paragraph* breakpoints set with the *bpg* (*breakpoint paragraph*) or *tpg* (*trace paragraph*) commands. Breakpoints set with other commands will remain in effect.
`tpg`	{ `tpg` / `trace paragraph` } [*command-list*]	Sets permanent *paragraph trace* breakpoints at the first executable statement of every HP COBOL II paragraph and section for which debugger information is available. The breakpoints are encountered each time the paragraph or section is entered. The command list, if any, is associated with the entry breakpoint. If no command list is specified, the entry command list defaults to: `{Q;t 2;c}`

Auxiliary Breakpoint Commands

Table I-13. Auxiliary Breakpoint Commands

Cmd	Syntax	Description
"*any string*"	"*any string*"	The *string* command displays any string that is enclosed in quotation marks.
`i`	$\left\{\begin{array}{l}\texttt{i}\\\texttt{if}\end{array}\right\}$ *expr command-list* [*command-list*]	The *i* (*if*) command lets you conditionally execute commands in a command list. If the expression evaluates to a non-zero value, the first group of commands is executed. If the expression evaluates to zero, the second command list, if it exists, is executed.
`Q`	$\left\{\begin{array}{l}\texttt{Q}\\\texttt{Quiet}\end{array}\right\}$	The *Q*(*Quiet*) command suppresses the "breakpoint at *address* ... " debugger messages that are normally displayed when a breakpoint is encountered. The *Q* (*Quiet*) command must be the first command in a command list; otherwise, it is ignored.

Assertion Control Commands

Table I-14. Assertion Control Commands

Cmd	Syntax	Description
`a`	$\left\{\begin{array}{l}\texttt{a}\\\texttt{assert}\end{array}\right\}$ *command-list*	Creates an assertion consisting of the command list that you enter. You can enclose the command list in braces to separate it from other commands on the same line.
`aa`	$\left\{\begin{array}{l}\texttt{aa}\\\texttt{activate assertion}\end{array}\right\}$ $\left[\begin{array}{l}\textit{number}\\*\end{array}\right]$	Activates the assertion having the number (ID) that you enter. Using the * option causes all assertions to be activated. Overall assertion mode is activated if the last suspended assertion is activated.
`da`	$\left\{\begin{array}{l}\texttt{da}\\\texttt{delete assertion}\end{array}\right\}$ $\left[\begin{array}{l}\textit{number}\\*\end{array}\right]$	Deletes the assertion having the number (ID) that you enter. Using the * option causes all assertions to be deleted.
`la`	$\left\{\begin{array}{l}\texttt{la}\\\texttt{list assertions}\end{array}\right\}$	Lists the number, the state (active or suspended) and the command list for each assertion, as well as the overall assertion state (active or suspended).
`sa`	$\left\{\begin{array}{l}\texttt{sa}\\\texttt{suspend assertion}\end{array}\right\}$ $\left[\begin{array}{l}\textit{number}\\*\end{array}\right]$	Suspends the assertion having the number that you enter. Using the * option causes all assertions to be suspended. Overall assertion mode is suspended if the last active assertion is suspended.
`ta`	$\left\{\begin{array}{l}\texttt{ta}\\\texttt{toggle assertions}\end{array}\right\}$	Toggles the overall assertion state between active and suspended.
`x`	$\left\{\begin{array}{l}\texttt{x}\\\texttt{exit}\end{array}\right\}$ $[\textit{expr}]$	Causes program execution to stop as if a breakpoint has been reached. If the expression (*expr*) is not given or it evaluates to zero, the debugger returns to command mode, ignoring any remaining commands in the assertion command list. If *expr* evaluates to non-zero, any remaining commands in the command list are executed.

Datatrace Control Commands

Table I-15. Datatrace Control Commands

Cmd	Syntax	Description
`ndt`	$\left\{\begin{array}{l}\texttt{ndt}\\ \texttt{datatrace}\end{array}\right\}\mathit{item}$ $\left[\left\{\mathit{command\text{-}list}\right\}\left[\texttt{silent}\right]\right]$	Creates a datatrace for the specified variable (*item*). You can enclose the command list in braces to separate it from other commands on the same line.
`adt`	$\left\{\begin{array}{l}\texttt{adt}\\ \texttt{activate datatrace}\end{array}\right\}$ $\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$	Activates the datatrace having the number (ID) that you enter. Using the * option causes all datatraces to be activated. Overall datatrace mode is activated if the last suspended data trace is activated.
`ddt`	$\left\{\begin{array}{l}\texttt{ddt}\\ \texttt{delete datatrace}\end{array}\right\}\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$	Deletes the datatrace having the number (ID) that you enter. Using the * option causes all datatraces to be deleted.
`ldt`	$\left\{\begin{array}{l}\texttt{ldt}\\ \texttt{list datatraces}\end{array}\right\}$	Lists the number, the state (active or suspended) and the command list for each datatrace, as well as the overall datatrace state (active or suspended).
`sdt`	$\left\{\begin{array}{l}\texttt{sdt}\\ \texttt{suspend datatrace}\end{array}\right\}\left[\begin{array}{l}\mathit{number}\\ *\end{array}\right]$	Suspends the datatrace having the number that you enter. Using the * option causes all datatraces to be suspended. Overall datatrace mode is suspended if the last active datatrace is suspended.
`tdt`	$\left\{\begin{array}{l}\texttt{tdt}\\ \texttt{toggle datatraces}\end{array}\right\}$	Toggles the overall datatrace state between active and suspended.
`x`	$\left\{\begin{array}{l}\texttt{x}\\ \texttt{exit}\end{array}\right\}\left[\mathit{expr}\right]$	Causes program execution to stop as if a breakpoint has been reached. If the expression (*expr*) is not given or it evaluates to zero, the debugger returns to command mode, ignoring any remaining commands in the datatrace command list. If *expr* evaluates to non-zero, any remaining commands in the command list are executed.

Record and Playback Commands

Table I-16. Record and Playback Commands

Cmd	Description
>*file*	Sets or changes the record file to *file*, turns recording on, rewrites the file from the beginning, and only records commands. If *file* exists, you are asked if you want to overwrite.
>>*file*	Sets or changes the record file to *file*, turns recording on, and only records commands. All recording is appended to the existing *file*; otherwise, a new file is created.
>	Displays the recording state and the current recording file. Can also use “>>”.
<*file*	Starts playback from the file.

<<*file*	Starts playback from the file using the “line-at-a-time” feature. Each command line from the playback file is shown before it is executed, and the debugger provides a list of the following commands for you to take some action:
	command (`<cr>`,`S`, `<num>`, `C`, `Q`, or `?`):
	You can use any of the above options as described:
	`<cr>` `execute one command line`
	`S` `skip one command line`
	`<num>` `execute number of command lines`
	`C` `continue through all playback`
	`Q` `quit playback mode`
	`?` `gives this explanation of the above commands`

`tr`	Toggles recording; toggles the state of the record mechanism between active and suspended.
>`t`	Turns recording on. (active)
>`f`	Turns recording off. (suspended)
>`c`	Closes the record file.

Table I-17. Commands Used to Record Debugger Output

Cmd	Description
>@*file*	Sets or changes the *record-all* file to *file*, rewrites from the beginning, and turns recording on. If *file* exists, you are asked if you want to overwrite. Captures all input to and output from the debugger command window, except user program output.
>>@*file*	Sets or changes the *record-all* file to *file*, and turns recording on. Appends *record-all* output to the existing *file*. Captures all input to and output from the debugger command window.
>`@`	Displays the current *record-all* state and file. Can also use ">>@".
`tr @`	Toggles the state of the *record-all* mechanism between active and suspended.
>`@t`	Turns *record-all* on.
>`@f`	Turns *record-all* off.
>`@c`	Closes the *record-all* file.

Macro Facility Commands

Table I-18. Macro Facility Commands

Cmd	Syntax	Description
`def`	`def` *name* *replacement-text*	Defines a macro substitution (user-defined command) for HP Symbolic Debugger commands. *Name* can be any string of letters or digits, beginning with a letter. *Replacement-text* can be any string of letters, blanks, tabs or other printing characters. The string must be contained on one line.
`tm`	$\left\{\begin{array}{l}\texttt{tm}\\ \texttt{toggle macros}\end{array}\right\}$	Toggles the state of the macro mechanism between active and suspended.
`undef`	`undef` $\left\{\begin{array}{l}\textit{name}\\ *\end{array}\right\}$	Removes macro defined as *name*. Using the * option causes all macros to become undefined.

Miscellaneous Commands

Table I-19. Miscellaneous Commands

Cmd	Syntax	Description
!	! [*MPE command*]	Escapes out of the debugger into the operating system. If a command is specified, it is automatically executed. Otherwise, a session is invoked and must be explicitly ended before the debugger can resume. When you execute the *!* command interactively, return to the debugger by hitting the (RETURN) key. When you use this command in an assertion or breakpoint command list, control returns to the debugger automatically. If you use the escape without giving a list of commands, you can return to the debugger by typing `exit` at the colon prompt.
:	: [*MPE command*]	Escapes out of the debugger into the operating system. If a command is specified, it is automatically executed. Otherwise, a session is invoked and must be explicitly ended before the debugger can resume. When you execute the *:* command interactively, return to the debugger by hitting the (RETURN) key. When you use this command in an assertion or breakpoint command list, control returns to the debugger automatically. If you use the escape without giving a list of commands, you can return to the debugger by typing `exit` at the colon prompt.
#	# [*text*]	Causes the text to be interpreted as a comment. The number symbol (#) must be the first nonblank character on the line.
(RETURN)	(RETURN)	Repeats the previous command. You can use this command with the following commands: ■ + ■ - ■ p (print) v (view)

Table I-19. Miscellaneous Commands (continued)

Cmd	Syntax	Description
`~`	`~`	Repeats the previous command. You must use the RETURN key after typing the ~. You can use this command with the following commands: ■ + ■ - ■ p (print) ■ v (view) ■ s (step) ■ S (Step)
`am`	{ `am` / `activate more` }	Activates (enables) the *more* feature.
`debug`	`debug`	Transfers control to the MPE NMdebugger by causing the child process to call the "DEBUG" entry point. When you exit, control returns to the HP Symbolic Debugger.
`f`	{ `f` / `format` } ["*printf-style-format*"]	Sets the printing format used by the debugger to print an address. Only the first 19 literal and formatting characters are used. (See the section on *printf* in the *HP C/XL Library Reference Manual* for a discussion of valid formats). Using the *f* (*format*) command without an argument will reset the format to the default format: 8 hexadecimal digits, preceded by "0x".
`g`	{ `g` / `goto` } { *line* / *#label* }	Moves the current point of execution suspension to the specified line or label. The specified line or label must be within the same procedure (or HP COBOL II paragraph) where execution is currently suspended (at depth zero on the stack). The program counter will change so that the given line number or the line that *#label* appears on becomes the next executable line. Execution does not automatically resume.

Table I-19. Miscellaneous Commands (continued)

Cmd	Syntax	Description
`h`	{ `h` | `help` } [*command*]	Prints a command summary, called the *Help* file which describes the syntax and use of each command. This facility references the short form of the command only, not the long form. The *more* facility can be used to view the file.
`q`	{ `q` | `quit` }	Quits the debugger and asks for confirmation: enter `y` (`yes`) or `n` (`no`).
`sm`	{ `sm` | `suspend more` }	Suspends the more feature and lets you view the output in a continuous stream.
`tc`	{ `tc` | `toggle case` }	Toggles case sensitivity; determines whether or not searches or names are case sensitive.
`do`	{ `do` } [*cmdid*] [`,` *editstring*]	Re-execute the command identified by *cmdid* after applying *editstring*. The optional *cmdid* can be a positive or negative number or a string that will be searched in the history stack.
`redo`	{ `redo` } [*cmdid*] [`,` *editstring*]	Allows you to edit and re-execute the command identified by *cmdid*. The optional *cmdid* can be a positive or negative number or a string that will be searched in the history stack.
`listredo`	{ `listredo` } [*start*] [`,` *end*]	Lists commands from the redo stack between *start* and *end* inclusive. The optional *start* and *end* can be positive or negative numbers.

J

Registers Displayed by HP Symbolic Debugger in Disassembly Mode

This appendix lists the registers displayed by HP Symbolic Debugger in disassembly mode.

Register Names

The actual register names used by the debugger as special variables are:

`$r0..$r31`	General registers
`$f0..$f15`	Floating-point (double-word) registers
`$pc`	Program counter
`$sp`	
`$dp`	
`$arg0..$arg3`	
`$ret0..$ret1`	

Registers Displayed by HP Symbolic Debugger in Disassembly Mode

The registers (or register fields) displayed by the debugger in disassembly mode are listed below. The first section lists the registers displayed in the *General Register and Floating-Point Register Window*. The second section lists the registers (or register fields) displayed in the *Special Register Window*.

Registers Displayed in the General and Floating-Point Register Window

`r0..r31`	General registers
`f0..f15`	Floating-point (double-word) registers
`pc`	(8x8) IASQ-head,IAOQ-head
`priv`	Privilege level, IAOQ[30..31]
`psw`	Process status word
`sar`	Shift amount register, CR11[27..31]

Registers Displayed in the Special Register Window

`tr0..7`	Temporary Registers, CR24..CR31
`sr0..7`	Space Registers
`pid1..4`	Protection Id's, CR8,9,12,13
`ccr`	Coprocessor configuration register, CR10
`sar`	Shift amount register, CR11
`eiem`	External interrupt enable mask, CR15
`itmr`	Internal Timer, CR16
`isr`	Interrupt Space Register, CR20
`iva`	Interrupt vector address, CR14
`rctr`	Recovery counter, CR0
`eirr`	External interrupt request register, CR23
`ior`	Interrupt offset register, CR21
`iir`	Interrupt instruction register, CR19
`pch`	(4x8) IASQ-head,IAOQ-head
`pct`	(4x8) IASQ-tail,IAOQ-tail
`priv`	Privilege level, IAOQ[30..31]
`psw`	Process status word

Glossary

address

Virtual memory address used to reference program code or data. When used to designate an address with the *ba* (*breakpoint address*) command, it can be either one of the following:

- Strictly a numeric value (such as 0x00001358)
- A symbolic address with or without an offset (such as main+0x1c).

assertion

A list of commands performed before the debugger executes a program statement. Useful for tracking unexpected changes in program data (undesired side effects).

breakpoint

A software "trigger" inserted into the user program, that, when encountered during execution, pauses the program and transfers back to the debugger. A breakpoint is always associated with a particular address, which is either specified explicitly or implied by its association with a line number, procedure entry or exit point.

Breakpoints can have the following associated with them:

- `Command list`- list of commands executed when the breakpoint is triggered
- `count`- how many times the breakpoint must be encountered before it is triggered.
- `lifespan`- "temporary" or "permanent" status (this information is actually determined by whether count is less than or greater than zero, respectively). A temporary breakpoint is removed when it is triggered; a permanent breakpoint is not.

child process

A subordinate process that is initiated and closely controlled by the debugger (parent). This process is a running instance of the program being debugged.

command

Commands tell the HP Symbolic Debugger which functions to perform, and can be spelled out or abbreviated. The abbreviation for most commands is the first character of each word in the command name. Commands are separated with a semicolon within a command list. For more information, see chapter 4 *HP Symbolic Debugger Commands*.

command list

A sequence of one or more debugger commands separated by a semicolon (;). Some commands expect command-lists as arguments. Braces ({}) must sometimes be used to enclose command-lists. For more information, see the individual command listings in chapter 4 *HP Symbolic Debugger Commands*.

current location

The "point-of-interest" in the source as displayed in the source window. Many commands take this as a default location. The current location is not necessarily the current point of program suspension (where the program is currently paused.)

datatrace

A list of commands performed when the value of the specified variable changes. Useful for tracking unexpected changes in program data (undesired side effects).

debugger information

Name, type, source file, and source-line-to-address mapping information generated by the compiler for use by the debugger. This information can significantly increase the size of an executable file. All debugger information is preprocessed (and reduced in size) the first time the program is debugged. This might increase initial startup time, which will thereafter be significantly shorter.

depth

Number of levels back in the current procedure call chain (stack). Depth 0 is where execution is suspended. If procedure A calls B, procedure B calls C, and C is where the program is suspended, then B is at depth 1 and A is at depth 2. The *trace* (*t*) or *Trace* (*T*) commands displays the procedures and their depths on the stack.

exception

Either a hardware or software generated condition that causes the program to be asynchronously suspended or halted. Examples of these might be:

- user-generated (keyboard) interrupt
- floating-point overflow
- segmentation violation (invalid addressing operation)
- bus error (invalid memory access)

expression

A valid combination of data object names, language operators, and constant numeric values. Every expression is evaluated and reduced to a single value.

file

The name of a file.

format

Used with the debugger commands *p* (*print*) or *disp* (*display*) to describe how data will be accessed and displayed. A format consists of:

- an optional repetition count

- a formatting character
- an optional object size

The access and display operation is performed once for each repetition (default 1). The number of bytes in each object is determined by the given object size (default depends on the formatting character). The formatting character determines how each object is interpreted and printed. For example, to print four sequential 16-bit integers in octal, use the format `4o2` or `4os`.

line mode
Debugger user interface that does not use any special terminal functions. This must be used for terminals that do not support memory lock.

location
A unique position in the user program. It can be specified as a file name, procedure name, source line number, or combination of these. An address (see above) can also be used to specify a location for certain commands.

machine instruction
Presented to the user when debugging in disassembly mode. Actual instruction mnemonics and syntax are described in the *HP Precision Architecture and Instruction Reference Manual*.

macro
Simple form of command aliasing using text substitution. A macro can be used as a shorthand for one or more commands.

memory lock
A terminal feature that allows some upper portion of the terminal screen to remain constant while the remainder of the screen is scrolled. This feature is required by the debugger for its window-oriented interface. If memory lock is unavailable, the line-oriented interface (line mode) is used.

procedure
A procedure, function, subroutine, paragraph, or module name. Also a user program name.

registers
Precision Architecture hardware registers. These are directly accessible by the debugger through symbolic names (e.g. $pc). Many registers have special meaning; some cannot be modified by the debugger user. See the *HP Precision Architecture and Instruction Reference Manual* for a discussion on the use of each register. Actual modification of hardware registers should not normally be necessary while debugging. Correct program execution depends highly on registers and their contents.

sharable code
Executable code that can be mapped into the address space of more than one process. No process should attempt to modify shared code while it is actually being shared by two or more processes. The debugger modifies the code in order to insert

breakpoints, requiring that multiple debugging sessions cannot occur with the same executable file. Private copies must be made first.

source
Source text (files) used to compile the user program. These can be in any of the programming languages supported by the debugger.

source line
A single line of text in a source file, denoted by a line number. A source line might or might not contain actual executable statements. Conversely, more than one statement can occur on a single line.

special variables
Named variable (prefixed by `S`) local to the debugger. Many special variables are predefined by the debugger to have a unique meaning. For example, $*line* is always the current line number, and $*dp* is the data-pointer register (HP-PA general register 27).

User-defined special variables are also available. They are created when first referenced, and allow you to store and reference numeric variables independent of the program being debugged.

stack
Linear data structure maintained by the user program for management of local data and flow of control during procedure calls. Each sequential region on the stack embodies information about a particular procedure. The preceding region (frame) describes its caller. At any point during execution, a stack_trace_ (generated by the *T* (*Trace*) command) will display information contained in each stack frame; in particular, the values of all local variables.

string
Quoted sequence of arbitrary characters. Quotes can be single (') or double (") depending on the current language ($*lang*). Character escapes allow inclusion of control or other non-printing characters.

variable
A variable name.

window
Region of the terminal screen limited to displaying specific information. The debugger has at least three: the source, location, and command windows.

Index